C++11 FAQ 中文版

wizardforcel

Published with GitBook



目錄

介紹 0

C++11 FAQ中文版 - C++11 FAQ 1

Stroustrup先生关于中文版的授权许可邮件 2

Stroustrup先生关于C++11 FAQ的一些说明 3

关于C++11的一般性的问题 4

- 您是如何看待C++11的? 4.1
- 什么时候C++0x会成为一部正式的标准呢？ 4.2
- 编译器何时将会实现C++11标准呢？ 4.3
- 我们何时可以用到新的标准库文件？ 4.4
- C++0x将提供何种新的语言特性呢？ 4.5
- C++11会提供哪些新的标准库文件呢？ 4.6
- C++0x努力要达到的目标有哪些？ 4.7
- 指导标准委员会的具体设计目标是什么？ 4.8
- 在哪里可以找到标准委员会的报告？ 4.9
- 从哪里可以获得有关C++11的学术性和技术性的参考资料？ 4.10
- 还有哪些地方我可以读到关于 C++0x的资料？ 4.11
- 有关于C++11的视频吗？ 4.12
- C++0x难学吗? 4.13
- 标准委员会是如何运行的？ 4.14
- 谁在标准委员会里？ 4.15
- 实现者应以什么顺序提供C++11特性？ 4.16
- 将会是C++1x吗？ 4.17
- 标准中的"concepts"怎么了? 4.18
- 有你不喜欢的C++特性吗？ 4.19

关于独立的语言特性的问题 5

- __cplusplus宏 5.1
- alignment(对齐方式) 5.2
- 属性（Attributes） 5.3
- atomic_operations 5.4
- auto – 从初始化中推断数据类型 5.5

C99功能特性 5.6
枚举类——具有类域和强类型的枚举 5.7
carries_dependency 5.8
复制和重新抛出异常 5.9
常量表达式（constexpr） 5.10
decltype – 推断表达式的数据类型 5.11
控制默认函数——默认或者禁用 5.12
控制默认函数——移动(move)或者复制(copy) 5.13
委托构造函数（Delegating constructors） 5.14
并发性动态初始化和析构 5.15
noexcept – 阻止异常的传播与扩散 5.16
显式转换操作符 5.17
扩展整型 5.18
外部模板声明 5.19
序列for循环语句 5.20
返回值类型后置语法 5.21
类成员的内部初始化 5.22
继承的构造函数 5.23
初始化列表 5.24
内联命名空间 5.25
Lambda表达式 5.26
用作模板参数的局部类型 5.27
long long（长长整数类型） 5.28
内存模型 5.29
预防窄转换 5.30
nullptr——空指针标识 5.31
对重载(override)的控制: override 5.32
对重载(override)的控制：final 5.33
POD 5.34
原生字符串标识 5.35
右角括号 5.36
右值引用 5.37
Simple SFINAE rule 5.38
静态（编译期）断言 — static_assert 5.39

模板别名（正式的名称为"template typedef")	5.40
线程本地化存储 (thread_local)	5.41
unicode字符	5.42
统一初始化的语法和语义	5.43
（广义的）联合体	5.44
用户定义数据标识（User-defined literals）	5.45
可变参数模板（Variadic Templates）	5.46
关于标准库的问题	6
abandoning_a_process	6.1
算法方面的改进	6.2
array	6.3
async()	6.4
atomic_operations	6.5
条件变量（Condition variables）	6.6
标准库中容器方面的改进	6.7
std::function 和 std::bind	6.8
std::forward_list	6.9
std::future和std::promise	6.10
垃圾回收（应用程序二进制接口）	6.11
无序容器（unordered containers）	6.12
锁（locks）	6.13
metaprogramming（元编程）and type traits	6.14
互斥	6.15
随机数的产生	6.16
正则表达式（regular expressions）	6.17
具有作用域的内存分配器	6.18
共享资源的智能指针——shared_ptr	6.19
smart pointers	6.20
线程（thread）	6.21
时间工具程序	6.22
标准库中的元组（std::tuple）	6.23
unique_ptr	6.24
weak_ptr	6.25

system error 6.26

C++11 FAQ中文版

来源：有{間}客栈 - C++11 FAQ中文版

C++11 FAQ中文版 - C++11 FAQ

更新至英文版January 3, 2012。

译者前言：

经过C++标准委员会的不懈努力，最新的ISO C++标准C++11，也即是原来的C++0x，已经正式发布了。让我们欢迎C++11！

今天获得Stroustrup 先生的许可，开始翻译由他撰写和维护的C++11 FAQ 。我

觉得这是一件伟大而光荣的事情，但是我又觉得压力很大，因为我的英语水平很差劲，同时自己的C++水平也很有限，很害怕在翻译过程中出现什么错误，贻笑大方不要紧，而误人子弟就罪过大了。所以，我这里的翻译只

能算是抛砖引玉，如果你的英文很好，你可以直接阅读他的原文 。或者，你也可以参照两者进行阅读，我想一定会有更多的收获。

当然，我也非常欢迎大家指出翻译中的错误，或者是加入进来和我一起翻译这份文档，共同为C++11在中国的推广做一点事情。你可以通过chenlq at live.com联系到我。

对自己的翻译做一点说明：

- 在翻译的过程中，尽量遵照原文含义，可能有时候也会自己根据自己的理解加一点批注，希望可以帮助大家理解。
- 另外，虽然C++11刚刚公布，但是现在已经有很多编译器支持C++11中一些相对比较独立的特性，比如gcc以及它在Windows下的MinGW，Visual C++ 2010也部分支持，大家可以使用这三款编译器尝试这个文档中的部分例子。
- 在下面的目录中，已经翻译的问题链接到相应的中文文档，未翻译的问题则链接到英文原文。

感谢所有参与翻译的志愿者：interma，Chilli，张潇，dabaidu，Yibo Zhu，lianggang jiang，nivo，陈良乔

友情提示：因为网站编辑器的原因，部分示例代码中模板类的模板参数会发生丢失，请读者阅读时注意参考原文的代码，由此给大家造成的不便，深表歉意。

在这里 有一份Stroustrup先生关于C++11的访谈，可以帮助你从更高地角度把握整个C++11新标准，你应该阅读 一下。

最后，祝大家阅读愉快:)

目录

- C++11 FAQ中文版 - C++11 FAQ
- Stroustrup先生关于中文版的授权许可邮件
- Stroustrup先生关于C++11 FAQ的一些说明
- 关于C++11的一般性的问题
  - 您是如何看待C++11的?
  - 什么时候C++0x会成为一部正式的标准呢？
  - 编译器何时将会实现C++11标准呢？
  - 我们何时可以用到新的标准库文件？
  - C++0x将提供何种新的语言特性呢？
  - C++11会提供哪些新的标准库文件呢？
  - C++0x努力要达到的目标有哪些？
  - 指导标准委员会的具体设计目标是什么？
  - 在哪里可以找到标准委员会的报告？
  - 从哪里可以获得有关C++11的学术性和技术性的参考资料？
  - 还有哪些地方我可以读到关于 C++0x的资料？
  - 有关于C++11的视频吗？
  - C++0x难学吗?
  - 标准委员会是如何运行的？
  - 谁在标准委员会里？
  - 实现者应以什么顺序提供C++11特性？
  - 将会是C++1x吗？
  - 标准中的"concepts"怎么了?
  - 有你不喜欢的C++特性吗？
- 关于独立的语言特性的问题
  - __cplusplus宏
  - alignment(对齐方式)
  - 属性（Attributes）
  - atomic_operations
  - auto – 从初始化中推断数据类型
  - C99功能特性
  - 枚举类——具有类域和强类型的枚举
  - carries_dependency
  - 复制和重新抛出异常
  - 常量表达式（constexpr）
  - decltype – 推断表达式的数据类型
  - 控制默认函数——默认或者禁用
  - 控制默认函数——移动(move)或者复制(copy)
  - 委托构造函数（Delegating constructors）
  - 并发性动态初始化和析构
  - noexcept – 阻止异常的传播与扩散

- 显式转换操作符
- 扩展整型
- 外部模板声明
- 序列for循环语句
- 返回值类型后置语法
- 类成员的内部初始化
- 继承的构造函数
- 初始化列表
- 内联命名空间
- Lambda表达式
- 用作模板参数的局部类型
- long long（长长整数类型）
- 内存模型
- 预防窄转换
- nullptr——空指针标识
- 对重载(override)的控制: override
- 对重载(override)的控制：final
- POD
- 原生字符串标识
- 右角括号
- 右值引用
- Simple SFINAE rule
- 静态（编译期）断言 — static_assert
- 模板别名（正式的名称为"template typedef"）
- 线程本地化存储 (thread_local)
- unicode字符
- 统一初始化的语法和语义
- （广义的）联合体
- 用户定义数据标识（User-defined literals）
- 可变参数模板（Variadic Templates）

- 关于标准库的问题
  - abandoning_a_process
  - 算法方面的改进
  - array
  - async()
  - atomic_operations
  - 条件变量（Condition variables）
  - 标准库中容器方面的改进
  - std::function 和 std::bind
  - std::forward_list

- std::future和std::promise
- 垃圾回收（应用程序二进制接口）
- 无序容器（unordered containers）
- 锁（locks）
- metaprogramming（元编程）and type traits
- 互斥
- 随机数的产生
- 正则表达式（regular expressions）
- 具有作用域的内存分配器
- 共享资源的智能指针——shared_ptr
- smart pointers
- 线程（thread）
- 时间工具程序
- 标准库中的元组（std::tuple）
- unique_ptr
- weak_ptr
- system error

Stroustrup先生关于中文版的授权许可邮件

On 2/18/2011 8:03 PM, ChenLiangqiao wrote:

> Dear Mr Stroustrup,Just as you said, there will be a long time before we can read your new book, the 4th edition of the C++ Programming Language.And it will take another long time to translate it into Chinese. Instead, may i translate some parts of your C++0x FAQ into Chinese and post it on my blog? Then, many Chinese C++ Developers will learn and use the C++0x much more early.
>
> Certainly. I think that is a great idea and would be a great service to Chinese programmers. I have just a few requests:
>
> Keep what you translate in one document so that I (and others) can link to it, rather than scattering its pieces all over your blog (you can of course refer to it from your blog as your translations progresses).
>
> Insert a link back to my original
>
> Translate whole sections rather than parts of sections

Stroustrup先生关于C++11 FAQ的一些说明

这份文档由Bjarne Stroustrup进行编写并维护。任何建设性的意见，校正，引用和建议，都是欢迎的。目前，我正在努力让这份文档更加完善并进行一些参考的清理工作。

C++11是下一个国际标准组织ISO的C++标准。目前，已经有草案可供大家参考提出意见。提供意见。以前的（和目前的）标准通常被称为为C++98和C++03。C++98和C++03之间的差异很小并且太过技术化，不应当引起用户的关注。

最终的标准委员会草案已经于2010年3月由国家标准机构表决通过。在让所有反馈意见都得到处理并让ISO的官员们都满意之前，还有很多工作要做。在现阶段，任何功能（即使是很小的）都不要指望被添加进入标准或者从标准中移出。C++0x这个名字只是我和其他人使用之后留下的一个遗留物，我们原本希望它是C++08或C++09。然而，为了减少混淆，我会继续谈到即将到来的C++标准，它有着与我们在这里为C++0x定义的相同的功能特性。我们可以把x看成是一个十六进制数，就像‘B’，这样C++0x就成了C++11。（译注：C++0x是这个新标准的代称，等标准通过之后，这个标准很可能被称为C++11。再译注：已经被正式确定为C++11了。）

如果你曾经就C++0x提出过一些建议，请找你们国家的标准化组织，或者是任何的标准化组织，向他们提交你关C++0x的建议和意见。目前，这是唯一的提交意见和建议的途径，这样可以保证标准委员会不用处理来自不同途径的相似的意见和建议。请记住，标准委员会全部由志愿者组成，他们的时间是有限的。

所有关于C++11的官方文档都可以在ISO C++标准委员会的官方网站上找到。标准委员会的官方名字是SC22 WG21。

请注意：这份FAQ将在很长一段时间内都是处于建设状态。任何的意见，建议，问题，参考，更正都是欢迎的。

目的

这份C++11 FAQ的目的是：

- 通过对比前一个ISO C++标准，让读者对C++11的新功能特性（包括语言特性和标注库的新功能）有一个大致的了解
- 介绍ISO C++标准影响的领域(?)
- 从用户的角度介绍C++0x的新功能特性
- 为更加深入的学习和研究C++11的新功能特性提供参考资料
- 为了铭记那些为新标准作出贡献的人，他们很多都是为标准委员会撰写报告的人。这个新标准并不是由一个不露面的组织撰写的。

请记住，这份FAQ的目的并不是为了全面地讨论那些功能特性，也不是详细地解释如何使用这些特性。它的目的是为了提供一些简单的例子以展示C++0x提供给我们的新功能（加上一些参考资料）。我的理想是，不管这个特性有多么复杂，“每个特性最多一页”。而更详细的信息，可以从参考资料中获得。

关于C++11的一般性的问题

您是如何看待C++11的?

对于我来说，这是一个最最容易被问到的问题。它可能是被问到的次数最多的问题。让人吃惊的是，C++11就像一种新的编程语言：跟以前旧的C++不同，C++11的各个部分被更好地组合在一起，并且我找到了一种更加自然的高层次的编程方式，而且同样有很好的效率。如果你仅仅是将C++当作更好的C，或者是一种面向对象语言，那么你将错过其中非常精彩和关键的东西。C++11中的抽象机制将比以前更加灵活，并且更加经济实惠。就像古老的“咒语”一样：如果你的头脑中有一个想法或者对象，想要在程序中直接对其进行表现，那么，你需要对现实世界中的对象进行建模，并在代码中对其进行抽象。现在这一过程更加容易了：你的想法将直接对应成为枚举、对象、类（例如，对默认值进行控制）、类的继承（例如，继承的构造函数）、模板、别名、异常、循环、线程等。这将远远好于以前那种简单的“以一双鞋适应所有脚”的抽象机制。

我的理想是，使用编程语言的各个功能来帮助程序员从另外一个角度思考系统的设计和实现。我认为C++11可以做到这一点。并且，不仅仅是为了让C++程序员可以做到，还包括更多的习惯于其它编程语言的，在更广泛的领域内进行系统编程的程序员都可以做到这一点。

换句话说，我依然是一个乐观主义者。

什么时候C++0x会成为一部正式的标准呢？

就是此刻！

一份正式标准的初稿产生于2008年9月。近期（2010年3月），一份最终的标准委员会草案标准即将接受国家标准机构的投票表决。

我们知道，新标准的看起来更象一个modulo minor（它已经随着独立的功能特性而发生了改变 ）(?)。新标准很可能命名为C++11，但即使是简单的官方审批程序也可能使之成为C++12。就个人而言，当我需要区分之前版本的时候，我更喜欢简单地用C++和年度标识来标记，例如ARM C++，C++98和C++03。现在，我按照惯例，将继续使用C++0x作为下一个C++标准的名字。将'X'看做十六进制数吧，这样更好理解一些。

(翻译：Chilli)

编译器何时将会实现C++11标准呢？

目前，业界普遍使用的已经发布的编译器（例如，GCC C++, Clang C++,IBM C++, 和 Microsoft C++）已经实现了许多C++11的特性。例如，在发布编译器时，同时发布全部或者绝大多数的新标准库文件似乎非常普遍，并且十分受用户的欢迎。我希望越来越多的新特性会出现在每次的版本发布中。可能性最大的，相对独立的特性，像auto, lambda, 和 strongly typed enums，我们将最早看到。我不禁猜想，何时所有编译器都将完全支持C++11——毫无疑问，这将会需要数年的时间 ——但我注意到，每一个C++11的特性都已经被一些人实现过，所以编译器的开发者们是有可用的实现经验可以参考的。（译注：关于各个主流编译器对C++11的支持情况，可以参考这里）

你也可以通过下面的链接获得各个主流编译器对C++11的支持情况：

- 各大主流编译器支持情况比较
- GCC
- IBM
- Microsoft
- EDG
- Clang

（翻译：Chilli）

我们何时可以用到新的标准库文件？

目前，随着GCC、Clang和Microsoft的实现，新标准库文件的初始版本已经开始发布，并且在Boost库中已经有很多标准库的组件可用。（注：Boost库是一个可移植、开放源码的C++库，作为标准库的后备，是C++标准化进程的发动机之一。）

（翻译：Chilli）

C++0x将提供何种新的语言特性呢？

你当然不会仅仅因为别人的一个想法，就给语言添加一个特性。事实上，关于C++，基本上每一个最现代化的语言特性都有人向我建议过，试着想象一下，C99、C#、Java、Haskell、Lisp、Python还有Ada的扩展集会是个什么样子？（译注：如果想着把这些语言的特点都集合到C++上，那C++就是一个四不像了）我们想问题要想的更加深入些，记住，即使标准委员会表决认为某个旧特性是不好的，完全剔除掉也是不可行的：事实表明，用户会迫使每一个开发者在兼容选项下（或默认）继续提供过时甚至已被禁止的特性达几十年。

为了试着从洪水般的建议中选择合理的建议，我们设计了一套具体的设计目标。我们不应该完全依据设计目标(?)，而且它也不能完全的指导标准委员会的每个细节（而且依我所见也不可能完全）。

其结果就是，C++成为一种被大大改良过的抽象机制的语言。这个抽象的范围比起手工操作的专业代码，大大增加了，而且C + +可以优雅，灵活，零成本的表达出来。当我们提到“抽象”的时候，人们往往只是想到“分类”或“对象”。C++0x中远不止这些：用户自定义的类型可以清晰安全的表达出来，而且类型的范围已经随着初始化列表，统一初始化，模板别名，右值引用，默认的和删除函数(?)特性以及可变参数模板等特性而不断增长扩大。而有些特性则简化了它们的实现，比如auto, inherited constructors和 decltype。这些增强功能足以使C++0x像一种新的语言。

已被接受的语言功能的列表，请参阅功能列表 。

（翻译：Chilli）

C++11会提供哪些新的标准库文件呢？

我本来也是希望看到更多的标准库文件的。然而，（我改观是因为我）注意到标准库文件中定义的篇幅就占了超过75%的规范性文字（而且这些还不包括作为参考文献的C标准库文件）。虽然我们中的许多人也很希望看到更多的标准库文件，但是没人责备库工作组的懈怠。值得一提的是，C++98标准库已通过新语言特性的应用，如初始化列表，右值引用，可变参数模板 和constexpr（常量表达式） 而取得了显著改善。相比C++98，C++11更易使用，并且能够提供更高的性能。

如果想查看所有已经被接受的标准库文件的列表，可以访问这里 the library component list .。

（翻译：Chilli）

C++0x努力要达到的目标有哪些？

C++是一种偏向于系统编程的通用编程语言，所以它应该：

- 支持数据抽象
- 支持面向对象的编程
- 支持泛型编程

C++0x努力的总体目标是为了加强：

- 使C++成为一种适用于系统编程和创建程序库的更好的语言——也就是直接利用C++进行编程，而不是为某个特定的领域提供专门的开发语言。（例如，专门为数值计算或Windows的应用程序开发提供支持）。
- 使C++更容易教和学——增加一致性，加强安全性，为初学者提供相关的配套组件（让C++更容易学习和使用）（初学者总是比专家多的）。（译注：C++0x现在真的是更好用了）

自然，这是在非常严格的兼容性约束下完成的。虽然我们在C++0x中引入了很多新的关键词（例如：static assert，nullptr，还有constexpr），但是标准委员会也很少会破坏标准库中的已经让人非常满意的代码。(?)

你可以通过下面这些参考获得更多详细的信息：

- B. Stroustrup: What is C++0x? . CVu. Vol 21, Issues 4 and 5. 2009.
- B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006 . ACM HOPL-III. June 2007.
- B. Stroustrup: A History of C++: 1979-1991 . Proc ACM History of Programming Languages conference (HOPL-2). March 1993.
- B. Stroustrup: C and C++: Siblings . The C/C++ Users Journal. July 2002.

（翻译：Chilli）

指导标准委员会的具体设计目标是什么？

自然，涉及不同标准化的不同组织或个人都会有某些不同的目的，尤其是在细节和优先级方面。此外，详细的目标总是随时间的改变而变动的。请记住，委员会做不到认同每个人的意见本身也是件好事——志愿者们的资源还是非常有限的。然而，这里已经有一套在实际探讨中使用着的规范，以此来确定那种特性或是库文件可适当的用C++0x中：

- 保持稳定和兼容性——不要打破旧代码，而如果你非打破不可的话，不要静静的做（注：应该是让做点工作告知大家吧）。
- 重库文件而非语言拓展–这是一条委员会做得不太成功的理念，因为太多人更喜欢实实在在的语言特性（而不是库）
- 重一般性而非专业性——聚焦于改善抽象机制（类，模板等）。
- 要专家新手都支持——新手可以通过更好的库文件及更多的一般性规则得到帮助，而专家需要一般且有效的特性。
- 提升类安全——主要的措施是通过允许程序员以避免类型不安全的性能。
- 提高性能和直接与硬件工作的能力——使C++甚至更好的用于嵌入式系统编程和高性能计算。
- 与实际世界相符——考虑工具链，实施成本，转换问题，ABI问题，教学和学习等注意到整合性能（新的和旧的）使之结合工作是个关键——基本上大部分的工作都是。整体大于各部分之和。另一种看待详细目的的方式是观察使用领域和使用风格：
- 机械模型和一致性——为使用现代硬件提供更强的保障和更好的设施（如多核及柔软的连贯内存模型?）。例子如 thread ABI, thread-local storage, 和 atomics ABI。
- 泛型编程——GP也是C ++ 98取得的巨大成就，我们需要基于经验改进对其的支持。例子像 auto 和 template aliases。
- 系统编程 – 改善与硬件相近的编程（如低级别的嵌入式系统编程），提高效率。例子有constexpr, std::array, 和 eneralized PODs.
- 库建设 – 消除抽象机制的局限性，效率低和不规范。例子有inline namespace, inherited constructors, 和 rvalue references.

（翻译：Chilli）未整理

在哪里可以找到标准委员会的报告？

前往the papers section of the committee's website 。那里有相当多的细节说明。寻找 “issues lists” 或 “state of” 列表（如 , State of Evolution (July 2008) ）。主要的分组有：

- Core （ CWG ）——处理语言技术事件并公式化
- Evolution （ EWG ）——处理与语言功能建议及横跨语言 / 库边界的事件问题
- 库 （ LWG ）——处理库设备提案问题

以下是提出的 C++Ox 标准建议草案。

（翻译：Chilli）未整理

从哪里可以获得有关C++11的学术性和技术性的参考资料？

你可以从以下这些地方获得你想要的资料：

- Bjarne Stroustrup: Software Development for Infrastructure. Computer, vol. 45, no. 1, pp. 47-58, Jan. 2012, doi:10.1109/MC.2011.353. A video interview about that paper and video of a talk on a very similar topic (That's a 90 minute talk incl. Q&A).
- Saeed Amrollahi:

  Modern Programming in New Millenium: A Technical Survey on Outstanding features of C++0x.

  Computer Report (Gozaresh-e Computer), No.199, November 2011 (Mehr and Aban 1390), pages 60-82. (in Persian)

- Hans-J. Boehm and Sarita V. Adve: Foundations of the C++ concurrency memory model . ACM PLDI'08.

- Hans-J. Boehm: Threads Basic . Yet unpublished technical report. // Introductory.
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++ . OOPSLA'06, October 2006. // The concept design and implementation as it stood in 2006; it has improved since, though not sufficiently to save it .
- Douglas Gregor and Jaakko Jarvi: Variadic templates for C++0x . Journal of Object Technology, 7(2):31-51, February 2008.
- Jaakko Jarvi and John Freeman: Lambda functions for C++0x . ACM SAC '08.
- Jaakko Jarvi, Mat Marcus, and Jacob N. Smith: Programming with C++ Concepts . Science of Computer Programming, 2008. To appear.
- M. Paterno and W. E. Brown : Improving Standard C++ for the Physics Community . CHEP'04. // Much have been improved since then!
- Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X . ACM ISMM'09.
- Verity Stob: An unthinking programmer's guide to the new C++ — Raising the standard . The Register. May 2009. (Humor (I hope)).
- [N1781=05-0041] Bjarne Stroustrup: Rules of thumb for the design of C++0x.
- Bjarne Stroustrup: Evolving a language in and for the real world: C++ 1991-2006 . ACM HOPL-III. June 2007. (incl. slides and videos). // Covers the design aims of C++0x, the standards process, and the progress up until 2007.
- B. Stroustrup: What is C++0x? . CVu. Vol 21, Issues 4 and 5. 2009.

- Anthony Williams: Simpler Multithreading in C++0x . devx.com.

上述列表是不完整的，随着有些人出版一些新的作品，列表中的文章会过时。如果你发现有文章应该出现在这个列表中，但事实上它并没有出现，请及时联系我进行更新。并不是所有的文献都能随着标准及时更新。我会尽力保证内容的一致性。

（翻译：nivo）

还有哪些地方我可以读到关于 C++0x的资料？

随着标准的日趋完善，有关 C++0x 的资料会愈来愈多， C++ 的各种实现 （例如，GCC， Visual C++） 开始提供新的语言特性和库。下面是资源列表：

- 委员会的网址 .
- C++0x 草案 .
- the C++0x 维基百科 . 积极维护， 但明显不是委员会的成员
- 帮助页 GCC’s experimental implementation of C++0x features .

（翻译：nivo）

有关于C++11的视频吗？

希望大家知道这真的是一个FAQ，而不是一些列出的我个人喜欢的问题集；我不喜欢技术话题的视频，我发现视频容易分心，并且经常包括一些微小的口头上的技术错误。

- B. Stroustrup, H. Sutter, H-J. Boehm, A. Alexandrescu, S.T.Lavavej, Chandler Carruth, and Andrew Sutter:

  several talks and panels from the Going Native 2012 conference.

- Herb Sutter:

  Writing modern C++ code: how C++ has evolved over the years. September 2011.

- Herb Sutter:

  C++ and Beyond 2011: Herb Sutter – Why C++?. August 2011.

- Try Google videos.
- Lawrence Crowl:

  Lawrence Crowl on C++ Threads.

  in Sophia Antipolis, June 2008.

- Bjarne Stroustrup:

  The design of C++0x

  at U of Waterloo in 2007.

- Bjarne Stroustrup:

  Initialization at Google in 2007.

- Bjarne Stroustrup:

  C++0x — An overview.

  in Sophia Antipolis, June 2008.

- Lawrence Crowl:

  Threads.

- Roger Orr:

  C++0x. January 2008.

- Hans-Jurgen Boehm:

  Getting C++ Threads Right. December 2007.

（翻译：nivo）

C++0x难学吗？

虽然我们不能在删除大量代码的前提下从C++中移除任何有影响的特性，C++0x仍旧比C++98大，所以如果你想熟知每一个规则，学习C++0x将会是很困难的。有两个工具可以帮助我们简化学习过程（从学习者的角度而言）

- 一般化: 替换，也就是用C++0x所提供的新特性替换C++以前所使用的各种特性。 (例如， uniform initialization, inheriting constructors, 和 threads). (?)（译注：这一段不太理解，但是从给出的例子来看，大约是某些原来使用C++98实现起来非常复杂的功能，现在可以在C++0x中轻松简便地实现，所以用C++0x替换C++98，比如线程就是一个非常明显的例子。）
- 简单化：提供比原来的方法更加简单的第二种选择。 (例如，array, auto, range for statement, and regex，这些特性都使得C++的开发更加简单。)

显然，“自下而上”的教/学方式将使得这些优势毫无发挥的地方，并且目前几乎没有别的不同方式。这应该随时间而变化。

（翻译：nivo）

标准委员会是如何运行的？

ISO标准委员会，SC22 WG21，是在ISO规则下运行的。奇怪的是，这些规则并非标准化的，而是随着时间的变化而变化。（译注：标准委员会的规则并不标准）

大多数国家都有活跃的C++团体并形成了自己的国家标准。这些团体举行会议，通过网络协调一致，并向ISO会议推选代表。加拿大，法国，德国，瑞士，英国和美国是出席这些会议较多的国家。丹麦，荷兰，日本，挪威，西班牙和别的一些国家则是出席人数比较少的国家。

大多数通过网络召开的会议都是介于这两者之间，会议记录由标准委员会编号并存放在WG21。

标准委员会每年召开两到三次会议，每次约一周的时间。这些会议的大部分工作就是工作划分，比如“核心“，”库“，”演化“，“并发”。根据需要，也会为了解决一些迫切的问题而召开会议，比如“概念”和“内存模型”。会议主要用来投票表决。首先，工作组举行民意投票来判断某个论点是否可以作为一个整体提案递交给标准委员会。然后，如果这个提案被接受，标准委员会将进行每人一票的投票表决。我们花费大量注意力在那些我们没有进入但是已经有很多人表现出来和国家不同意的形势——这将会导致长期的争论。最后，官方草案的投票由国家标准机构通过邮件完成。

标准委员会和C标准组织以及POSIX有正式的联系，并和其他一些组织也有或多或少的联系。

(翻译：nivo)

谁在标准委员会里？

标准委员会包括大约200个人，其中有大约60位会出席每年两到三次一周时间的会议。除此之外，在一些国家还有一些国家标准组织和会议。多数成员通过出席会议，邮件讨论或提交论文供标准委员会斟酌等方式贡献自己的力量。多数成员有朋友或同事提供帮助。第一天，标准委员会召集从各个国家而来的代表，并且每一次会议由6到12个国家的代表参加。最终投票由20个国家标准组织完成。这样，ISO C++标准是一个集合了众人集体智慧而成的最终成果，而并不是人们通常认为的——它只是一些为了创造完美语言的不相干的人创造出来的空中楼阁。这个标准需要获得他们的同意，只有这样才可保证所有人可以接受标准。

很自然地，多数志愿者（并非全部）有他们自己的日常工作：我们有的人开发编译器，有的人写生成工具，有的人写程序库，还有人写应用程序（此类人很少），还有少数的研究者，还有顾问，还有编写测试工具的等等。

这有一个简短的关于组织者的列表：Adobe,Apple,Boost,EDG,Google,HP,IBM,Intel,Microsoft,Red Hat,Sun.

这还有一个标准委员会的简短的成员列表，你有可能会在网上或是著作里遇到他们：

Dave Abrahams, Matt Austern, Pete Becker, Hans Boehm, Steve Clamage, Lawrence Crowl, Beman Dawes, Doug Gregor, Howard Hinnant, Jaakko Jarvi, Francis Glassborow, Jens Maurer, Jason Merrill, Sean Parent, P.J. Plauger, Tom Plum, Gabriel Dos Reis, Bjarne Stroustrup, Herb Sutter, David Vandervoorde Michael Wong. Apologies 还有更多成员就不一一列出了。请关注一些论文的作者列表：一个标准是由很多人共同完成的，而不是一个匿名的标准委员会。

你可以从WG21 papers 获得有关这些作者的更深入的专长介绍以获得更深的印象，但请牢记那些为标准的完成做出了巨大贡献但并没有写太多东西的人们。

（翻译：nivo）

实现者应以什么顺序提供C++11特性？

标准中并没有关于引入C++0x特性的顺序；它只是简单的列出了为了达到完整地C++11特性所需要做的事情。然而，如果实现者分阶段引入新的C++0x特性，我们也认为这是合理的。毕竟，我不会使用不支持的特性。所以，一个基于"易于提供"和"对多数人有用"的理念，是早期实现的关键原则。

- 没有功能特性的新库取决于新的语言特性，比如可变参数模板和常量表达式constexpr。
- 简单且易于实现的特性将在细小但重要的地方帮助用户：
- auto
- enum class枚举类
- long long
- nullptr空指针
- right brackets右括号
- static_assert静态断言

帮助实现C++11 标准库的语言特点：

- 常量表达式
- 初始化列表
- 一般的和统一的初始化 (包括 预防宽转窄)
- 右值引用
- 可变参数模板
- 标准库用到的所有特点

相关的并发特性：

- memory model 内存模型
- 线程的本地化存储thread_local
- atomic types 原子类型
- local types as template arguments 作为模板参数的局部类型
- lambdas
- 标准库的完整支持
- PODs

如果你看得仔细，你会发现我对很多语言特点（在引入这些特性的时间上）并没有看法。很自然地，我也希望这些特性能够被尽快地实现。但是，我并没有一个关于何时这些特性应该被实现的判断。显然，每一个C++实现者都有自己的原则，所以我们不能期望他们步调一致，但是我希望他们可以稍微关注一下别人在做什么，这样可以让用户更早地开始他们的移植工作。

（翻译：nivo）

将会是C++1x吗？

几乎可以肯定，并且不仅仅是因为标准委员会要拖延C++11的期限。我听到最多的希望或计划是社区应该在C++11推出后立即开始实施。与当今科技发展水平相比，标准的发布周期太长了，所以一些人认为三年时间用来修订比较合适。而我认为5年则是更为现实的。那C++16呢？

（翻译：nivo）

标准中的"concepts"怎么了？

概念（concept）（译注：这里翻译并不准确，请大家参照原文）是允许精确地描述模板参数的一个特性，不幸的是，社区认为未来的关于概念的工作会严重影响标准的进度，并从工作文件中移出了这个特性，相关解释可以参阅我的笔记：移出概念的决定和 一个基于概念的DevX 观点和C++的启示。

即使概念将来成为后续C++的一部分，我也不得不从该文档中删除这一章节，但是把它们放在后面：

- 公理 (语义假设)
- 概念
- 概念图 （concepts map）

(翻译：nivo)

有你不喜欢的C++特性吗？

是的，有些C++98中的特性我是不喜欢的，比如宏。问题在于，并非是我喜欢什么或者我发现它对我需要做得一些事有帮助。事实上，这个问题是，无论是否有人认为确实需要说服他人支持这个想法，或者一些用法在某些用户社区已经根深蒂固到必须提供支持的地步。

（翻译：nivo）

关于独立的语言特性的问题

__cplusplus宏

在C++11中，__cplusplus宏将被设定为一个比以往的标准中的值（在C++03中，是199711L）更大的值。

alignment(对齐方式)

属性（Attributes）

“属性”是C11标准中的新语法，用于让程序员在代码中提供额外信息。相较于风格各异的传统方式(**attribute**, __declspec, #pragma等)，“属性”语法致力于将各种“方言”进行统一。

与传统语法不同的是，“属性”语法相当灵活，可以随处添加，且总是作用于与之相邻的语法实体。

```
void f [[ noreturn ]] () // f() 永不返回
{
  throw "error"; // 虽然不得返回，但可以抛出异常
}

struct foo* f [[carries_dependency]] (int i); // 编译优化指示
int* g(int* x, int* y [[carries_dependency]]);
```

正如你看到的那样，属性被放置在两个双重中括号“[[…]]”之间。目前，noreturn和carries_dependency是C++11标准中仅有的两个通用属性。

我们有理由担心属性的大量使用会引起C++语言的混乱，很可能将产生很多C++语言的“方言”。

所以，我们推荐仅在不影响源代码的业务逻辑的前提下，才使用属性来帮助编译器作更好的错误检查（例如，[[noreturn]]，或者是帮助代码优化（例如， [[carries_dependency]]）。

在未来的计划中，属性的一个重要用途是为OpenMP提供更好的辅助信息。例如：

```
// 使用[[omp::parallel()]]属性告诉编译器，这个for循环可以并行执行
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) {
// ...
```

就像上面的代码展示的那样，通过指定for循环的[[omp::parallel()]]属性，编译器将使用OpenMP对这个for循环进行并行化处理，从而这个for循环将并行执行。

参考文献:

Standard: 7.6.1 Attribute syntax and semantics, 7.6.3-4 noreturn, carries_dependency 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions

[N2418=07-027] Jens Maurer, Michael Wong: Towards support for attributes in C++ (Revision 3)

atomic_operations

Stroustrup尚未完成此主题，期待中

对此主题感兴趣的朋友，可以参考

C++小品：榨干性能：C++11中的原子操作（atomic operation）

VC11有点甜：原子操作和`< atomic>`头文件

参考：

auto – 从初始化中推断数据类型

考虑下面的代码：

```
auto x = 7;
```

这里的变量x被整数7初始化，所以x的实际数据类型是int。auto的通用形式如下：

```
auto x = expression;
```

这样，这个表达式计算结果的数据类型就是变量x的数据类型。

当数据类型未知或不便书写时，使用auto可让编译器自动根据用以初始化变量的表达式类型来推导变量类型。考虑如下代码：

```
template<class T> void printall(const vector<T>& v)
{
      // 根据v.begin()的返回值类型自动推断p的数据类型
      for (auto p = v.begin(); p!=v.end(); ++p)
          cout << *p << “n”;
}
```

为了表示同样的意义，在C++98中，我们不得不这样写：

```
template<class T> void printall(const vector<T>& v)
{
    for (typename vector<T>::const_iterator p = v.begin();
       p!=v.end(); ++p)
        cout << *p << “n”;
}
```

当变量的数据类型依赖于模板参数时，如果不使用auto关键字，将很难确定变量的数据类型。例如：

```
template<class T,classs U> void multiply (const vector<T>& vt,
const vector<U>& vu)
{
    // …
    auto tmp = vt[i]*vu[i];
    // …
}
```

在这里，tmp的数据类型应该与模板参数T和U相乘之后的结果的数据类型相同。对于程序员来说，要通过模板参数确定tmp的数据类型是一件很困难的事情。但是，对于编译器来说，一旦确定了T和U的数据类型，推断tmp的数据类型将是轻而易举的一件事情。

auto特性是C++11中最早被提出并被实现的特性。早在1984年，我就在我的Cfont中实现了auto特性，但是由于一些兼容性问题，它没有被纳入以前的标准。当C++98和C99同意删除“implicit int”之后，这些兼容性问题已经不复存在了，也就是C++语言对“每个变量和函数都要有确切的数据类型”的要求消失了。auto关键字原来的含义（表示local变量）是多余而无用的——标准委员会的成员们在数百万行代码中仅仅只找到几百个用到auto关键字的地方，并且大多数出现在测试代码中，有的甚至就是一个bug。

auto主要用于简化代码，因此并不会影响标准库规范。

参考：

- the C++ draft section 7.1.6.2, 7.1.6.4, 8.3.5 (for return types)
- [N1984=06-0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis:Deducing the type of variable from its initializer expression (revision 4).

C99功能特性

为了与C语言标准保持高度的兼容性，在C标准委员会的协助之下，一些细小的改变被引入到C++0x中。

- long long
- 扩展的整型数据类型（例如，关于可选的更长的整型数的规则）
- 关于UCN的改变[N2170==07-0030]: 解除了”字符常量/字面字符串中不得使用控制/基本的通用字符名”的限制

```
// 译注: C++03中允许通过\uNNNN的形式
// 在字符/字符串中引入非ASCII字符(Unicode)
// 但是控制字符以及基本的英文字母、数字、符号等
// 小于0x00A0的字符(与ASCII兼容)不在此列。
// 而在C++11中，这一限制被进一步放开
// OK in C++03 and C++11
const char* str1 = “Hello \u3366”;
// Err in C++03, OK in C++11
const char* str2 = “Hello \u0033”;
```

- 宽/窄字符串的连接
- Not VLAs（变长数组）

添加了一些扩展的预处理规则

- **func** a该宏经过宏展开(宏替换)后即为当前函数名
- **STDC_HOSTED**
- _Pragma: _Pragma(X) 扩展成#pragma X
- 支持可变长度参数的宏（通过不同数目的参数对宏进行重载）

  ```
  #define report(test, …) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
  ```

- 空的宏参数

标准库中的很多功能组件都是继承自C99（从本质上来说，所有C99的库的改变都是从C89继承而来的）

参考:

Standard: 16.3 Macro replacement.

[N1568=04-0008] P.J. Plauger: PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99.

枚举类——具有类域和强类型的枚举

枚举类（“新的枚举”/“强类型的枚举”）主要用来解决传统的C++枚举的三个问题：

- 传统C++枚举会被隐式转换为int，这在那些不应被转换为int的情况下可能导致错误
- 传统C++枚举的每一枚举值在其作用域范围内都是可见的，容易导致名称冲突(同名冲突)
- 不可以指定枚举的底层数据类型，这可能会导致代码不容易理解、兼容性问题以及不可以进行前向声明

枚举类（enum）（“强类型枚举”）是强类型的，并且具有类域：

```
enum Alert { green, yellow, election, red }; // 传统枚举
 enum class Color { red, blue };   // 新的具有类域和强类型的枚举类
 // 它的枚举值在类的外部是不可直接访问的，需加“类名::”
 //  不会被隐式转换成int
 enum class TrafficLight { red, yellow, green };
 Alert a = 7;   //  错误，传统枚举不是强类型的，a没有数据类型
 Color c = 7;   // 错误，没有int到Color的隐式转换
 int a2 = red;           // 正确，Alert被隐式转换成int
// 在 C++98中是错误的，但是在C++11中正确的
int a3 = Alert::red;
int a4 = blue;            // 错误，blue并没有在类域中
int a5 = Color::blue; // 错误，没有Color到int的默认转换
Color a6 = Color::blue;   // 正确
```

正如上面的代码所展示的那样，传统的枚举可以照常工作，但是你现在可以通过提供一个类名来改善枚举的使用，使其成为一个强类型的具有类域的枚举。

因为新的具有类名的枚举具有传统的枚举的功能（命名的枚举值），同时又具有了类的一些特点（枚举值作用域处于类域之内且不会被隐式类型转换成int），所以我们将其称为枚举类（enum class）。

因为可以指定枚举的底层数据类型，所以可以进行简单的互通操作以及保证枚举值所占的字节大小：

```
enum class Color : char { red, blue }; // 紧凑型表示(一个字节)
// 默认情况下，枚举值的底层数据类型为int
enum class TrafficLight { red, yellow, green };
// E占几个字节呢？旧规则只能告诉你：取决于编译器实现
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };

// C11中我们可以指定枚举值的底层数据类型大小
enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
```

同时，由于能够指定枚举值的底层数据类型，所以前向声明得以成为可能：

（译注：就是在枚举类定义之前就使用这个枚举类的名字声明指针或引用变量）

```
enum class Color_code : char;     // (前向) 声明
void foobar(Color_code* p);   // 使用
// ...
// 定义
enum class Color_code : char { red, yellow, green, blue };
```

枚举类的底层数据类型必须是有符号或无符号的整型，默认情况下是int。

标准库中也使用了枚举类。

- 系统特定的错误代码，定义在`<system_error>: enum class errc</system_error>`
- 指针安全指示，定义在`<memory>: enum class pointer_safety { relaxed, preferred, strict }</memory>`
- I/O流错误，定义在`<iosfwd>: enum class io_errc { stream = 1 }</iosfwd>`
- 异步通信错误，定义在`<future>: enum class future_errc { broken_promise, future_already_retrieved, promise`

其中的某些枚举类还定义了操作符重载，比如“==”等。

参考：

the C++ draft section 7.2

[N1513=03-0096] David E. Miller: Improving Enumeration Types (original enum proposal).

[N2347 = J16/07-0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: Strongly Typed Enums (revision 3).

[N2499=08-0009] Alberto Ganesh Barbati: Forward declaration of enumerations.

carries_dependency

复制和重新抛出异常

你怎么捕获一个异常，之后在另外一个线程上重新抛出？使用在标准文档18.8.5中描述的异常传递中的方法吧，那将显示标准库的魔力。

exception_ptr current_exception(); 返回一个exception_ptr 变量，它将指向现在正在处理的异常（15.3）或者现在正在处理的异常的副本（拷贝），或者有的时候在当前没有遇到异常的时候，返回值为一个空的exception_ptr变量。只要exception_ptr指向一个异常，那么至少在exception_ptr的生存期内，运行时能够保证被指向的异常是有效的。

```
void rethrow_exception(exception_ptr p);
template exception_ptr copy_exception(E e);
```

它的作用如同：

```
try {
    throw e;
} catch(...) {
    return current_exception();
}
```

当我们需要将异常从一个线程传递到另外一个线程时，这个方法十分有用.

常量表达式（constexpr）

常量表达式机制是为了：

- 提供一种更加通用的常量表达式
- 允许用户自定义的类型成为常量表达式
- 提供了一种保证在编译期完成初始化的方法（可以在编译时期执行某些函数调用）

考虑下面这段代码：

```
enum Flags { good=0, fail=1, bad=2, eof=4 };
constexpr int operator|(Flags f1, Flags f2)
{ return Flags(int(f1)|int(f2)); }
void f(Flags x)
{
    switch (x) {
    case bad:         /* … */ break;
    case eof:         /* … */ break;
    case bad|eof:     /* … */ break;
    default:          /* … */ break;
    }
}
```

在这里，常量表达式关键字constexpr表示这个重载的操作符“|”只应包含形式简单的运算，如果它的参数本身就是常量，那么这个操作符应该在编译时期就应该计算出它的结果来。（译注：我们都知道，switch的分支条件要求常量，而使用constexpr关键字重载操作符“|”之后，虽然“bad|eof”是一个表达式，但是因为这两个参数都是常量，在编译时期，就可以计算出它的结果，因而也可以作为常量对待。）

除了可以在编译时期被动地计算表达式的值之外，我们希望能够强制要求表达式在编译时期计算其结果值，从而用作其它用途，比如对某个变量进行赋值。当我们在变量声明前加上constexpr关键字之后，可以实现这一功能，当然，它也同时会让这个变量成为常量。

```
    constexpr int x1 = bad|eof;    // ok
    void f(Flags f3)
    {
        // 错误：因为f3不是常量，所以无法在编译时期计算这个表达式的结果值
        constexpr int x2 = bad|f3;
        int x3 = bad|f3;     // ok，可以在运行时计算
    }
```

使用constexpr强制在运行期求值，一般用于全局对象（或namespace内的对象），尤其是那些放在只读区的对象。

除了基本类型外，对于那些构造函数比较简单的对象和由其构成的表达式，也可以成为常量表达式

```
struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x(xx), y(yy){}
};
constexpr Point origo(0,0);
constexpr int z = origo.x;
constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexpr int x = a[1].x;    // x 变成常量1
```

需要注意的是，constexpr并不是const的通用版，反之亦然：

- const主要用于表达“对接口的写权限控制”，即“对于被const修饰的量名(例如const指针变量)，不得通过它对所指对象作任何修改”。(但是可以通过其他接口修改该对象)。另外，把对象声明为const也为编译器提供了潜在的优化可能。具体来说就是，如果把一个量声明为const，并且没有其他地方对该量作取址运算，那么编译器通常(取决于编译期实现)会用该量的实际常量值直接替换掉代码中所有引用该量的地方，而不用在最终编译结果中生成对该量的存取指令。
- constexpr的主要功能则是让更多的运算可以在编译期完成，并能保证表达式在语义上是类型安全的。(译注：相比之下，C语言中#define只能提供简单的文本替换，而不具任何类型检查能力)。与const相比，被constexpr修饰的对象则强制要求其初始化表达式能够在编译期完成计算。之后所有引用该常量对象的地方，若非必要，一律用计算出来的常量值替换。

（译注：zwvista的一段评论，有助于我们理解constexpr的意义，感谢zwvista。constexpr 将编译期常量概念延伸至括用户自定义常量以及常量函数，其值的不可修改性由编译器保证，因而constexpr 表达式是一般化的，受保证的常量表达式。）

参考：

- the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The constexpr specifier
- [N1521=03-0104] Gabriel Dos Reis: Generalized Constant Expressions (original proposal).
- [N2235=07-0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: Generalized Constant Expressions — Revision 5 .

(翻译：陈良乔，感谢：zwvista)

decltype – 推断表达式的数据类型

decltype(E)是一个标识符或者表达式的推断数据类型(“declared type”)，它可以用在变量声明中作为变量的数据类型。例如：

```
void f(const vector<int>& a, vector<float>& b)
{
    // 推断表达式a[0]*b[0]的数据类型，并将其定义为Temp类型
    typedef decltype(a[0]*b[0]) Tmp;
    // 使用Tmp作为数据类型声明变量，创建对象

    for (int i=0; i < b.size(); ++i) {
      Tmp* p = new Tmp(a[i]*b[i]);
      // …
   }
   // …
}
```

这个想法以“typeof”的形式已经在通用语言中流行很久了，但是，现在使用中的typeof的实现并没有完成，并有一些兼容性问题，所以新标准将其命名为decltype。

如果你仅仅是想根据初始化值为一个变量推断合适的数据类型，那么使用auto是一个更加简单的选择。当你只有需要推断某个表达式的数据类型，例如某个函数调用表达式的计算结果的数据类型，而不是某个变量的数据类型时，你才真正需要delctype。

参考

? the C++ draft 7.1.6.2 Simple type specifiers

? [Str02] Bjarne Stroustrup. Draft proposal for “typeof”. C++ reflector message c++std-ext-5364, October 2002. (original suggestion).

? [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto (original proposal).

? [N2343=07-0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Decltype (revision 7): proposed wording.

控制默认函数——默认或者禁用

的复制构造函数和赋值操作符的情况下，便编译器会为我们生成默认的复制构造函数和赋值操作符，以内存复制的形式完成对象的复制。虽然这种机制可以为我们节省很多编写复制构造函数和赋值操作符的时间，但是在某些情况下，比如我们不希望对象被复制，这种机制却是多此一举。）

关于类的“禁止复制”，现在可以使用delete关键字完美地直接表达：

```
class X {
    // …
    X& operator=(const X&) = delete; // 禁用类的赋值操作符
    X(const X&) = delete;
};
```

相反地，我们也可以使用default关键字，显式地指明我们希望使用默认的复制操作：

```
class Y {
    // …
    // 使用默认的赋值操作符和复制构造函数
    Y& operator=(const Y&) = default; // 默认的复制操作
    Y(const Y&) = default;
};
```

显式地使用default关键字声明使用类的默认行为，对于编译器来说明显是多余的，但是对于代码的阅读者来说，使用default显式地定义复制操作，则意味着这个复制操作就是一个普通的默认的复制操作。 将默认的操作留给编译器去实现将更加简单，更少的错误发生 ，并且通常会产生更好的目标代码。

“default”关键字可以用在任何的默认函数中，而“delete”则可以用于修饰任何函数。例如，我们可以通过下面的方式排除一个不想要的函数参数类型转换：

```
struct Z {
    // …
    Z(long long); // 可以通过long long初始化
    Z(long) = delete; // 但是不能将long long转换为long进行初始化(?)
};
```

参考：

- the C++ draft section ???
- [N1717==04-0157] Francis Glassborow and Lois Goldthwaite: explicit class and default definitions (an early proposal).
- Bjarne Stroustrup: Control of class defaults (a dead end).
- [N2326 = 07-0186] Lawrence Crowl: Defaulted and Deleted Functions .

控制默认函数——移动(move)或者复制(copy)

委托构造函数（Delegating constructors）

在C++98中，如果你想让两个构造函数完成相似的事情，可以写两个大段代码相同的构造函数，或者是另外定义一个init()函数，让两个构造函数都调用这个init()函数。例如：

```
class X {
    int a;
    // 实现一个初始化函数
    validate(int x) {
        if (0<x && x<=max) a=x; else throw bad_X(x);
    }
public:
    // 三个构造函数都调用validate()，完成初始化工作
    X(int x) { validate(x); }
    X() { validate(42); }
    X(string s) {
        int x = lexical_cast<int>(s); validate(x);
    }
    // …
};
```

这样的实现方式重复罗嗦，并且容易出错。并且，这两种方式的可维护性都很差。所以，在C++0x中，我们可以在定义一个构造函数时调用另外一个构造函数：

```
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    // 构造函数X()调用构造函数X(int x)
    X() :X{42} { }
    // 构造函数X(string s)调用构造函数X(int x)
    X(string s) :X{lexical_cast<int>(s)} { }
    // …
};
```

（译注：在一个构造函数中调用另外一个构造函数，这就是委托的意味，不同的构造函数自己负责处理自己的不同情况，把最基本的构造工作委托给某个基础构造函数完成，实现分工协作。）

参考：

- the C++ draft section 12.6.2
- N1986==06-0056 Herb Sutter and Francis Glassborow: Delegating Constructors (revision 3).

并发性动态初始化和析构

（译注：这部分作者还没有完成，不过一旦英文版出来，中文版将进行同步更新，请读者多多关注！）

参考：

- [N2660 = 08-0170] Lawrence Crowl: Dynamic Initialization and Destruction with Concurrency (Final proposal).

（翻译：lianggang jiang）

noexcept – 阻止异常的传播与扩散

如果一个函数不能抛出异常，或者一个程序并没有接获某个函数所抛出的异常并进行处理，那么这个函数可以用新的noexcept关键字对其进行修饰，表示这个函数不会抛出异常或者抛出的异常不会被接获并处理。例如：

```
extern "C" double sqrt(double) noexcept;    // 永远不会抛出异常

vector my_computation(const vector& v) noexcept // 在这里，我不准备处理内存耗尽的异常，所以我只是
{
    vector res(v.size());    // 可能会抛出异常
    for(int i; i         return res;
}
```

如果一个经过noexcept修饰的函数抛出异常（异常会尝试逃出这个函数（？）），程序会通过调用terminate()来结束执行。通过 terminate()的调用来结束程序的执行会带来很多问题，例如，无法保证对象的析构函数的正常调用，无法保证栈的自动释放，同时也无法在没有遇到任 何问题的情况下重新启动程序。所以，它是不可靠的。

我们这样写是故意的，它使得成为一种简单、粗暴但是非常有效的机制（比那种旧的动态地抛出异常的机制要有效得多）。

同时，我们还可以让一个函数根据不同的条件实现noexcept修饰或者是无noexcept修饰。例如，一个算法可以根据它用作模板参数所使用的操作是否抛出异常，来决定自己是否抛出异常。例如：

```
template
void do_f(vector& v) noexcept(noexcept(f(v.at(0)))); //如果f(v.at(0))可以抛出异常，则这个函数也
{
    for(int i; i             v.at(i) = f(v.at(i));
}
```

在这里，第一个noexcept被用作操作符（operator）：如果if f(v.at(0))不能够抛出异常，noexcept(f(v.at(0)))则返回true，也即意味着f()和at()是无法抛出异常（noexcept）。

noexcept()操作符是一个常量表达式，并且不计算表达式的值，只是判断这个表达式是否会产生并抛出异常。

声明的通常形式是noexcept(expression)，并且单独的一个“noexcept”关键字实际上就是的一个noexcept(true)的简化。一个函数的所有声明都必须与noexcept声明保持 兼容。

一个析构函数不应该抛出异常；通常，如果一个类的所有成员都拥有noexcept修饰的析构函数，那么这个类的析构函数就自动地隐式地noexcept声明，而与函数体内的代码没有关系。

通常，将某个抛出的异常进行移动操作是一个很坏的主意，所以，在任何可能的地方都用noexcept进行声明。如果某个类的所有成员都有使用 noexcept声明的析构函数，那么这个类默认生成的复制或者移动操作（类的复制构造函数，移动构造函数等）都是隐式的noexcept声明。（？）

noexcept 被广泛地系统地应用在C++11的标准库中，以此来提供标准库的性能和满足标准库对于简洁性的需求。

参考：

- Standard: 15.4 Exception specifications [except.spec].
- Standard: 5.3.7 noexcept operator [expr.unary.noexcept].
- [N3103==10-0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept must terminate, as it does).
- [N3167==10-0157] David Svoboda: Delete operators default to noexcept .
- [N3204==10-0194] Jens Maurer: Deducing "noexcept" for destructors
- [N3050==10-0040] D. Abrahams, R. Sharoni, and D. Gregor: Allowing Move Constructors to Throw (Rev. 1) .

显式转换操作符

C++98标准提供隐式和显式两种构造函数，也就是说，声明为显式形式的构造函数所定义的转换只能用于显式转换，而其他形式的构造函数则用于隐式转换。例如：

```
struct S { S(int); };     // “普通构造函数”默认是隐式转换
S s1(1);         // ok, 直接构造
S s2 = 1;    // ok, 隐式拷贝构造
void f(S);
// 能通过编译（但是经常会产生意外结果——如果S是vector类型会怎么样呢？）
// 译注：详见下一用例的解释
f(1);

struct E { explicit E(int); };    // 显式构造函数
E e1(1);         // ok
E e2 = 1;    // 错误（但是常常会让人感到意外——这怎么会错呢？）
void f(E);
// 该处会产生编译错误（而非编译通过），以避免因隐式类型转换而得到莫名其妙的结果。
// 例如std::vector::vector(int size), 该构造函数在标准库中定义为显式类型转换,
// （译注：以避免程序员为了初始化一个只含有一个元素10的数组而写出如下代码:
//     vector<int> vec = 10;
//    而实际上该代码的含义却是定义一个初始包含10个元素的数组）
f(1);
```

然而，禁止从构造函数作隐式转换（以避免问题），并没有堵住全部漏洞。如果某个类本身禁止改动，那么可以从另一个不同的类中定义一个转换操作符。例如：

```
struct S { S(int) { } /* … */ };
struct SS {>
    int m;
    SS(int x) :m(x) { }
    // 在struct S无须定义S(SS)——所谓的“非侵入”式做法
    operator S() { return S(m); }
};
SS ss(1);    // ok; 默认构造函数
S s1 = ss;   // ok; 隐式转换为S后调用拷贝构造函数
S s2(ss);    // ok; 隐式转换为S后调用直接构造函数
void f(S);
f(ss);        // ok; 隐式转换为S后传参
```

（译注：这段代码的意义，实际是通过SS作为中间桥梁，将int转换为S。）

遗憾的是，C++98中无法定义”显式转换操作符”来完全禁止某个类相关的隐式转换（因为除此之外鲜有用武之地）。C++11则高瞻远瞩，添加了这个特性。例如：

```
struct S { S(int) { } };
struct SS {
    int m;
    SS(int x) :m(x) { }
    // 因为结构体S中没有定义构造函数S(SS)
    // 无法将SS转换为S，所以只好在SS中定义一个返回S的转换操作符，
    // 将自己转换为S。
    // 转换动作，可以由目标类型S提供，也可以由源类型SS提供。）
     explicit operator S() { return S(m); }
};
SS ss(1);    // ok; 默认构造函数
S s1 = ss;    // 错误; 拷贝构造函数不能使用显式转换
S s2(ss);    // ok; 直接构造函数可以使用显式转换
void f(S);
f(ss);        // 错误; 从SS向S的转换必须是显式的.
// 译注: 强制类型转换也可使用显式转换，例如
// S s3 = static_cast<S>(ss);
```

参考:

- Standard: 12.3 Conversions
- [N2333=07-0193] Lois Goldthwaite, Michael Wong, and Jens Maurer:

  Explicit Conversion Operator (Revision 1).

扩展整型

如若引入扩展整型，到时将有一整套的规则说明应该如何进行整型扩展（及其精度）。

// 译注: 作稿之时，扩展整型尚未确定是否纳入C++11

参见：

- [06-0058==N1988] J. Stephen Adamczyk: Adding extended integer types to C++ (Revision 1) .

（翻译：lianggang jiang）

外部模板声明

模板特化可以被显式声明，这可以作为消除多重实例化的一种方式。例如：

```
#include "MyVector.h"
extern template class MyVector<int>; // 消除下面的隐式实例化
// MyVector 类将在“其他地方”被程序员显式实例化
void foo(MyVector<int>& v)
{
    //在这个地方使用vector类型
}
```

下列代码就是上例中的”其他地方”：

```
#include "MyVector.h"
// 使MyVector类对客户端（clients）可用（例如，共享库）
template class MyVector<int>;
```

这种方法的主要目的是为避免编译器和链接器的大量“去除冗余的实例化”工作。

参见：

- Standard 14.7.2 Explicit instantiation
- [N1448==03-0031] Mat Marcus and Gabriel Dos Reis: Controlling Implicit Template Instantiation

  .

（翻译：lianggang jiang）

序列for循环语句

C++11中引入了序列for循环以实现区间遍历的简化。这里所谓的区间可以是任一种能用迭代器遍历的区间，例如STL中由begin()和end()定义的序列。所有的标准容器，例如std::string、初始化列表、数组，甚至是istream，只要定义了begin()和end()就行。

这里是一个序列for循环语句的例子：

```
void f(const vector& v)
{
    for (auto x : v) cout << x << 'n';
    for (auto& x : v) ++x;    // 使用引用，方便我们修改容器中的数据
}
```

可以这样理解这里的序列for循环语句，"对于v中的所有数据元素x"，循环由v.begin()开始，循环到v.end()结束。又如：

```
for (const auto x : { 1,2,3,5,8,13,21,34 })
   cout << x << 'n';
```

begin()函数（包括end()函数）可以是成员函数通过x.begin()方式调用，或者是独立函数通过begin(x)方式调用。

（译注：好像C#中早就有这种形式的for循环语句，用于遍历一个容器中的所有数据很方便，难道C++是从C#中借用过来的？）

或参见：

- the C++ draft section 6.5.4 (note: changed not to use concepts)
- [N2243==07-0103] Thorsten Ottosen:

  Wording for range-based for-loop (revision 2).

- [N3257=11-0027] Jonathan Wakely and Bjarne Stroustrup: Range-based for statements and ADL (Option 5 was chosen).

返回值类型后置语法

考虑下面这段代码：

```
template<class T, class U>
??? mul(T x, U y)
{
    return x*y;
}
```

函数mul()的返回类型要怎么写呢？当然，是“xy类型”，但是这并不是一个数据类型，我们如何才能一开始就得到它的真实数据类型呢？在初步了解*C++0x*之后，你可能一开始想到使用*decltype*来推断“xy”的数据类型：

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // 注意这里的作用域
{
    return x*y;
}
```

但是，这种方式是行不通的，因为x和y不在作用域内。但是，我们可以这样写：

```
template<class T, class U>
// 难看别扭，且容易产生错误
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)
{
    return x*y;
}
```

如果称这种用法为“还可以”，就已经是过誉了。

C++11的解决办法是将返回类型放在它所属的函数名的后面：

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{
    return x*y;
}
```

这里我们使用了auto关键字，（auto在C++11中还有根据初始值推导数据类型的意义），在这里它的意思变为“返回类型将会稍后引出或指定”。

返回值后置语法最初并不是用于模板和返回值类型推导的，它实际是用于解决作用域问题的。

```
struct List {
    struct Link { /* ... */ };
    Link* erase(Link* p);   // 移除p并返回p之前的链接
    // ...
};

List::Link* List::erase(Link* p) { /* ... */ }
```

第一个List::是必需的，这仅是因为List的作用域直到第二个List::才有效。更好的表示方式是：

```
auto List::erase(Link* p) -> Link* { /* ... */ }
```

现在，将函数返回类型后置，Link*就不需要使用明确的List::进行限定了。

参考：

- the C++ draft section ???
- [Str02] Bjarne Stroustrup. Draft proposal for “typeof”. C++ reflector message c++std-ext-5364, October 2002.
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek:

  Decltype and auto.

- [N2445=07-0315] Jason Merrill:

  New Function Declarator Syntax Wording.

- [N2825=09-0015] Lawrence Crowl and Alisdair Meredith:

  Unified Function Syntax.

类成员的内部初始化

在C++98标准里，只有static const声明的整型成员能在类内部初始化，并且初始化值必须是常量表达式。这些限制确保了初始化操作可以在编译时期进行。例如：

```
int var = 7;
class X {
    static const int m1 = 7;   // 正确
    const int m2 = 7;    // 错误：无static
    static int m3 = 7;              // 错误：无const
    static const int m4 = var;  // 错误：初始化值不是常量表达式
    static const string m5 = “odd”; //错误：非整型
    // …
};
```

C++11的基本思想是，允许非静态（non-static）数据成员在其声明处（在其所属类内部）进行初始化。这样，在运行时，需要初始值时构造函数可以使用这个初始值。考虑下面的代码：

```
class A {
public:
    int a = 7;
};
```

这等同于：

```
class A {
public:
    int a;
    A() : a(7) {}
};
```

单纯从代码来看，这样只是省去了一些文字输入，其实它的真正永无之地在于拥有多个构造函数的类。因为大多情况下，对于同一个成员，多个构造函数应使用相同的值去初始化。例如：

```
class A {
public:
     A(): a(7), b(5), hash_algorithm(“MD5"),
       s(“Constructor run”) {}
    A(int a_val) :
      a(a_val), b(5), hash_algorithm(“MD5"),
      s(“Constructor run”)
      {}
    A(D d) : a(7), b(g(d)),
        hash_algorithm(“MD5"), s(“Constructor run”)
        {}
    int a, b;
private:
    // 哈希加密函数可应用于类A的所有实例
    HashingFunction hash_algorithm;
    std::string s;  // 用以指明对象正处于生命周期内何种状态的字符串
};
```

对于每一个构造函数，程序员必须使用完全一样的字面值来来初始化hash_algorithm和s这两个成员。但是并不是所有人都记得严格遵守这条规则，一旦出现纰漏，程序将难以维护。C++11给出了解决之道：可在成员声明的地方直接赋以初值：

```
class A {
public:
    A(): a(7), b(5) {}
    A(int a_val) : a(a_val), b(5) {}
    A(D d) : a(7), b(g(d)) {}
    int a, b;
private:
    //哈希加密函数可应用于类A的所有实例
    HashingFunction hash_algorithm{“MD5"};
    //用以指明对象正处于生命周期内何种状态的字符串
    std::string s{“Constructor run”};
};
```

如果一个成员同时在类内部初始化时和构造函数内被初始化，则只有构造函数的初始化有效（这个初始化值“优先于”默认值）（译注：可以认为，类内部初始化先于构造函数初始化进行，如果是对同一个变量进行初始化，构造函数初始化会覆盖类内部初始化）。因此，我们可以进一步简化：

```
class A {
public:
    A() {}
    A(int a_val) : a(a_val) {}
    A(D d) : b(g(d)) {}
    int a = 7;
    int b = 5;
private:
    //哈希加密函数可应用于类A的所有实例
    HashingFunction hash_algorithm{“MD5"};
    //用以指明对象正处于生命周期内何种状态的字符串
    std::string s{“Constructor run”};
};
```

参考文献：

- the C++ draft section “one or two words all over the place”; see proposal.

- [N2628=08-0138] Michael Spertus and Bill Seymour:

  Non-static data member initializers.

（翻译：lianggang jiang）

继承的构造函数

人们有时会对类成员函数或成员变量的作用域问题感到困惑，尤其是，当基类与派生类的同名成员不在同一个作用域内时：

```
struct B {
    void f(double);
};
struct D : B {
    void f(int);
};
B b;   b.f(4.5);    // OK
// 调用的到底是B::f(doube)还是D::f(int)呢？
// 实际情况往往会让人感到意外：调用的f(int)函数实参为4
D d;   d.f(4.5);
```

在C++98标准里，可以将普通的重载函数从基类“晋级”到派生类里来解决这个问题：

```
struct B {
    void f(double);
};

struct D : B {
    using B::f;     // 将类B中的f()函数引入到类D的作用域内
    void f(int);    // 增加一个新的f()函数
};

B b;   b.f(4.5);    // OK
// 可行：调用类D中的f(double)函数
// 也即类B中的f(double)函数
D d;   d.f(4.5);
```

普通重载函数可以通过这种方式解决，那么，对于构造函数又该怎么办呢？我曾经说过“不能像应用于普通成员函数那样，将上述语法应用于构造函数，这如历史偶然一样”。为了解决构造函数的“晋级”问题，C++11提供了这种能力：

```
class Derived : public Base {
public:
    // 提升Base类的f函数到Derived类的作用范围内
    // 这一特性已存在于C++98标准内
    using Base::f;
    void f(char);     // 提供一个新的f函数
    void f(int);  // 与Base类的f(int)函数相比更常用到这个f函数
    // 提升Base类的构造函数到Derived的作用范围内
    // 这一特性只存在于C++11标准内
    using Base::Base;
    Derived(char);    // 提供一个新的构造函数
    // 与Base类的构造函数Base(int)相比
    // 更常用到这个构造函数
    Derived(int);
    // …
};
```

如果这样用了，仍然可能困惑于派生类中继承的构造函数，这个派生类中定义的新成员变量需要初始化（译注：基类并不知道派生类的新增成员变量，当然不会对其进行初始化。）：

```
struct B1 {
    B1(int) { }
};
struct D1 : B1 {
    using B1::B1; // 隐式声明构造函数D1(int)
    int x;
};
void test()
{
    D1 d(6);  // 糟糕：调用的是基类的构造函数，d.x没有初始化
    D1 e;    // 错误：类D1没有默认的构造函数
}
```

我们可以通过使用成员初始化（member-initializer）消除以上的困惑：

```
struct D1 : B1 {
    using B1::B1;    // 隐式声明构造函数D1(int)
    // 注意：x变量已经被初始化
    // （译注：在声明的时候就提供初始化）
    int x{0};
};
void test()
{
    D1 d(6);    // d.x的值是0
}
```

参考：

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

  Initialization and initializers

  (an overview of initialization-related problems with suggested solutions).

- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

  Initializer lists.

- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

  Initializer lists (Rev. 3) .

- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

  Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

初始化列表

考虑如下代码：

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string,string>> languages = {
    {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};
map<vector<string>,vector<int>> years = {
    { {"Maurice","Vincent", "Wilkes"},
              {1913, 1945, 1951, 1967, 2000} },
    { {"Martin", "Ritchards"},
              {1982, 2003, 2007} },
    { {"David", "John", "Wheeler"},
              {1927, 1947, 1951, 2004} }
};
```

现在，初始化列表不再仅限于数组。可以接受一个“{}列表”对变量进行初始化的机制实际上是通过一个可以接受参数类型为std::initializer_list的函数（通常为构造函数）来实现的。例如：

```
void f(initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({});  // 以空列表为参数调用f()
f{1,2}; // 错误：缺少函数调用符号( )
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

初始化列表可以是任意长度，但必须是同质的（所有的元素必须属于某一模板类型T, 或可转化至T类型的)。

容器可以用如下方式来实现“初始化列表构造函数”：

```
template<class E> class vector {
public:
    // 初始化列表构造函数
            vector (std::initializer_list<E> s)
    {
           // 预留出合适的容量
              reserve(s.size());    //
        // 初始化所有元素
            uninitialized_copy(s.begin(), s.end(), elem);
     sz = s.size(); // 设置容器的size
    }
    // ... 其他部分保持不变 ...
};
```

使用“{}初始化”时，直接构造与拷贝构造之间仍有细微差异，但不再像以前那样明显。例如，std::vector拥有一个参数类型为int的显式构造函数及一个带有初始化列表的构造函数：

```
vector<double> v1(7); // OK: v1有7个元素<br />
v1 = 9;          // Err: 无法将int转换为vector
vector<double> v2 = 9;    // Err: 无法将int转换为vector

void f(const vector<double>&);
f(9);                // Err: 无法将int转换为vector

vector<double> v1{7};     // OK: v1有一个元素，其值为7.0
v1 = {9};            // OK: v1有一个元素，其值为9.0
vector<double> v2 = {9};  // OK: v2有一个元素，其值为9.0
f({9});  // OK: f函数将以列表{9}为参数被调用

vector<vector<double>> vs = {
    vector<double>(10), // OK, 显式构造(10个元素，都是默认值0.0)
    vector<double>{10}, // OK：显式构造(1个元素，值为10.0)
        10           // Err ：vector的构造函数是显式的
};
```

函数可以将initializer_list作为一个不可变的序列进行读取。例如：

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

仅具有一个std::initializer_list的单参数构造函数被称为初始化列表构造函数。

标准库容器，string类型及正则表达式均具有初始化列表构造函数，以及（初始化列表）赋值函数等。初始化列表亦可作为一种“序列”以供“序列化for语句”使用。（译注：参见“序列for循环语句”章节）

初始化列表同时也是“统一初始化”方案的一部分。（译注：参见“统一初始化的语法和语义”章节）

参考：

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

  Initialization and initializers

  (an overview of initialization-related problems with suggested solutions).

- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

  Initializer lists.

- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

  Initializer lists (Rev. 3) .

- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

（翻译：dabaitu）

内联命名空间

内联命名空间旨在通过"版本"的概念，来实现库的演化。考虑如下代码：

```
// 文件：V99.h
 inline namespace V99 {
         void f(int);     // 对V98版本进行改进
         void f(double);  // 新特性
         // …
 }
 // 文件：V98.h
 namespace V98 {
     void f(int);        // V98版本只实现基本功能
     // …
 }
 // 文件：Mine.h
 namespace Mine {
 #include “V99.h”
 #include “V98.h”
 }
```

上述命名空间Mine中同时包含了较新的版本(V99)以及早期的版本(V98)，如果你需要显式使用（某个版本的函数），你可以：

```
#include “Mine.h”
using namespace Mine;
// …
V98::f(1);        // 早期版本
V99::f(1);        // 较新版本
f(1);            // 默认版本
```

此处的要点在于，被inline修饰的内联命名空间，其内部所包含的所有类/函数/变量等声明，看起来就好像是直接在外围的命名空间中进行声明的一样。（译注：我们注意到，这里的f(1)函数调用相当于显式调用Mine::V99::f(1)，使用inline关键字定义的内联名字空间成为默认名字空间。（就像内联函数一样，内联的名字空间被嵌入到它的外围名字空间，成为外围名字空间的一部分。）

inline描述符是一个非常"静态(static)"及面向实现的设施，它由库的设计者选择在（某个版本namespace之前）放置，且一旦选定则库的所有使用者只能被动接受（译注：即命名空间的作者可以通过放置inline描述符来表示当前最新的命名空间是哪个，所以对用户来说，这个选择是"静态"的：用户无权判断哪个命名空间是最新的）。因此，Mine命名空间的用户没法选择说："我想要默认的命名空间为V98，而非V99"。

参考：

- Standard 7.3.1 Namespace definition [7]-[9].

（翻译：dabaitu）

Lambda表达式

（译注:目前支持lambda的gcc编译器版本为4.5，其它详细的编译器对于C++11新特性的支持请参考http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport）

Lambda表达式是一种描述函数对象的机制，它的主要应用是描述某些具有简单行为的函数（译注：Lambda表达式也可以称为匿名函数，具有复杂行为的函数可以采用命名函数对象，当然,何谓复杂，何谓简单，这取决于编程人员的个人选择）。例如：

```
vector<int> v = {50, -10, 20, -30};
std::sort(v.begin(), v.end());    // 排序时按照默认规则
// 此时v中的数据应该是 { -30, -10, 20, 50 }

// 利用Lambda表达式，按照绝对值排序
std::sort(v.begin(), v.end(), [](int a, int b)
{ return abs(a)<abs(b); });
// 此时v应该是 { -10, 20, -30, 50 }
```

参数 `[](int a, int b) { return abs(a) < abs(b); }` 是一个具有如下行为的"lambda"：接受两个整数a和b，然后返回对它们的绝对值进行" `<` "比较的结果。

Lambda表达式可以访问在它被调用的作用域内的局部变量。例如：

```
void f(vector<Record>& v)
{
    vector<int> indices(v.size());
    int count = 0;
    generate(indices.begin(),indices.end(),[&count]()
    { return count++; });

    // 对indices按照记录的名字域顺序进行排序
    std::sort(indices.begin(), indices.end(), [&](int a, int b)
    { return v[a].name<v[b].name; });
    // ...
}
```

有人认为这“相当简洁”，也有人认为这是一种可能产生危险且晦涩的代码的方式。我的看法是，两者都正确。

[&] 是一个“捕捉列表(capture list)”，用于描述将要被lambda函数以引用传参方式使用的局部变量。如果我们仅想“捕捉”参数v，则可以写为: [&v]。而如果我们想以传值方式使用参数v，则可以写为：[=v]。如果什么都不捕捉，则为：[]。将所有的变量以引用传递方式使用时采用[&], 而相对地，使用[=] 则相应地表示以传值方式使用所有变量。（译注：“所有变量”即指lambda表达式在被调用处，所能见到的所有局部变量）

如果某一函数的行为既不通用也不简单，那么我建议采用命名函数对象或者函数。例如，如上示例可重写为：

```
void f( vector<Record>& v)
{
    vector<int> indices(v.size() );
    int count = 0;
    fill(indices.begin(), indices.end(), [&]()
    { return ++count; };

    struct Cmp_names {
        const vector& vr;
        Cmp_names(const vector<Record>& r) : vr(r) {}
        bool operator() (Record& a, Record& b) const
                { return vr[a] < vr[b]; }
    };

    //对indices按照记录的名字域顺序进行排序
    std::sort(indices.begin(), indices.end(), Cmp_names(v) );
        }
```

(译注：此处采用了函数对象Cmp_names(v)来代替lambda表达式，由于Cmp_names具有以引用传参方式的构造函数，因此Cmp_names(v)相当于使用了”[&v]”的lambda表达式)

对于简单的函数功能，比如记录名称域的比较，采用函数对象就略显冗长，尽管它与lambda表达式生成的代码是一致的。在C++98中，这样的函数对象在被用作模板参数时必须是“非本地”的（译注：即你不能在函数对象中像此处的lambda表达式那样使用被调用处的局部变量），然而在C++中（译注：意指C++0x），这不再是必须的。

为了描述一个lambda，你必须提供：

- 它的捕捉列表：即（除了形参之外）它可以使用的变量列表（” `[&]` ” 在上面的记录比较例子中意味着“所有的局部变量都将按照引用的方式进行传递”）。如果不需要捕捉任何变量，则使用 []。
- （可选的）它的所有参数及其类型（例如： `(int a, int b)` ）。
- 组织成一个块的函数行为（例如： `{ return v[a].name < v[b].name; }` ）。
- （可选的）使用”返回值类型后置语法“来指明返回类型。但典型情况下，我们仅从return语句中去推断返回类型，如果没有返回任何值，则推断为void。

参考：

- Standard 5.1.2 Lambda expressions
- [N1968=06-0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine:

  Lambda expressions and closures for C++

  (original proposal with a different syntax)

- [N2550=08-0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl:

  Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4) (final proposal).

- [N2859=09-0049] Daveed Vandevoorde:

New wording for C++0x Lambdas.

用作模板参数的局部类型

在C++98中，局部类和未命名类不能作为模板参数，这或许是一个负担，C++11则放宽了这方面的限制：

```
void f(vector<X>& v)
{
    struct Less {
        bool operator()(const X& a, const X& b)
        { return a.v<b.v; }
    };
    // C++98: 错误: Less是局部类
        // C++11: 正确
        sort(v.begin(), v.end(), Less());
}
```

当然除了这里的局部类之外，在C++11中，我们还可以采用Lambda表达式来做同样的事情：

```
void f(vector<X>& v)
{
    sort(v.begin(), v.end(),
        [] (const X& a, const X& b) { return a.v<b.v; });
}
```

尽管如此，我们仍然不要忘记，为一系列动作行为命名有利于文档化，是一个值得鼓励的设计风格。而且，非局部的函数体（当然也需要命名）还可以被重用于其他模块。

C++11同时也允许模板参数使用未命名类型的值：

```
template<typename T> void foo(T const& t){}
enum X { x };
enum { y };

int main()
{
    foo(x);      // C++98: ok; C++11: ok
    //（译注：y是未命名类型的值，C++98无法从这样的值中推断出函数模板参数）
    foo(y);      // C++98: error; C++11: ok
    enum Z { z };
    foo(z);      // C++98: error; C++11: ok
    //（译注：C++98不支持从局部类型值推导模板参数
}
```

参考：

- Standard: Not yet: CWG issue 757
- [N2402=07-0262] Anthony Williams:

  Names, Linkage, and Templates (rev 2).

- [N2657] John Spicer:

Local and Unnamed Types as Template Arguments.

long long（长长整数类型）

这是一个至少为64 bit的整数类型（译注：实际宽度依赖于具体的实现平台），例如：

```
long long x = 9223372036854775807LL;
```

不过，不要想当然地认为存在long long long或者将long拼写为short long long。

（译注：如同J. Stephen Adamczyk在参考文献中所言，”long long”是一个晦涩的拼写64-bit整数类型的方式，也不是一个可以解决不断增长的数据类型宽度的有效方法，目前，它仅仅是一个用于表达64bit整形数的标准。）

参考：

- the C++ draft ???.
- [05-0071==N1811] J. Stephen Adamczyk:

  Adding the long long type to C++ (Revision 3).

内存模型

（译注：这一个item有相当深的理论深度，原文也比较晦涩难懂，翻译者提醒大家，最好参照原文理解，如果翻译中有什么不恰当的地方，还请批评指出，不胜感谢。）

所谓“内存模型”，是计算机（硬件）体系结构与编译器双方之间的一种约定。有了它，大多数程序员便不用处处考虑日新月异的计算机硬件细节。如果没有内存模型，那么线程机制、锁机制及无锁编程等都无从谈起。

内存模型的最关键保证是：两个线程可以各自独立地存取各自的“内存位置”而不会互相影响。那么，什么是“内存位置”呢？ 一个内存位置要么是一个标量类型的对象，要么是一个连续且位宽非零的最长比特域组。例如：此处的S具有四个独立的内存位置：

```
struct S {<br />
    char a;   // 位置#1
    int b:5,  // 位置#2
    int c:11,
    // 注意：":0"是一个“特殊符号”
    // (译注：此处的":0"会将一个int对象分割为两个内存位置，
    // 因为上面所讲的内存位置的第二种情况需要“连续且位宽非零”)
    int :0,
    int d :8;  // 位置#3
    struct {int ee:8; } e;  // 位置#4
};
```

内存模型为什么如此重要？为什么它不是显而易见的？(译注：此处指“为什么应用程序所使用的内存模型与计算机的物理内存结构不是直接一致的，即不用经过任何中间层”，在本例中，应当指为什么b和c属于同一个内存位置)。 难道并非一直如此吗？

问题在于，如果多个计算任务真正地并发运行，那么当这些来自不同计算任务中不相关（很明显）的指令代码在同一时刻执行时，内存硬件的诡异行为将会被暴露无遗（译注：一个诡异行为就是，在上述例子中，对变量b,c,d的存取，并非一次仅操作5,11或者8个bit。具体操作与处理器的行为有关）。事实上，如果没有了编译器的支持，指令流/数据流以及缓存命中等细节问题将会被直接暴露给应用程序开发人员，而他们对此毫无对策。即使两个线程之间不存在数据共享，情况依然糟糕如此！考虑如下两个分开编译的“线程”：

```
//线程 1:
char c;
c = 1;
int x = c;

//线程 2
char b;
b = 1;
int y = b;
```

为了尽量模拟现实状态，我使用了分开编译（对每个线程）来保证编译器/优化器不会对内存进行优化（译注：此处指对每个线程内的代码进行逐行编译，然后连接，以避免代码优化），以防止“聪明”的编译器跳过读取变量c和b而直接将x和y初始化为1。那么现在，x和y的值可能是什么呢？按照C++11的标准，唯一正确的答案，也是显而易见的那个，x和y均为1。但如果你采用了一个传统意义上的优秀的带预并发处理(pre-concurrency)的C或者C++编译器，那么事情变得有趣起来：x和y的可能值会是0和0(很少发生)，或者1和0，或者0 和1，或者1和1。这些都是在“真实环境”下观察到的情况。为何如此呢？原因在于，链接器可能会将变量c和b的位置分配在相邻的内存上（在同一个word内），而且90年代的C/C++标准并未对此作出禁令（译注：指不拒绝这种变量在内存中的存储位置安排方式）。在这种情况下，C++就如同所有那些未考虑实际硬件并发就进行设计的语言一样，由于现代CPU无法读写单个的字节，读写操作在处理中总是以word为单位进行的，所以，对变量c的赋值实际上是“读取包含变量c的一个word，替换掉其中的c部分，然后再写回这个word”。由于对变量b的赋值也是如此进行的，这就造成了两个操作变量b和变量c的线程有如此多地机会互相干涉、影响（译注：线程1操作变量c时，回写操作会改变变量b的值，线程2也是如此，这就造成了两个线程之间的干扰，而b和c也处于不稳定状态），即使它们之间并未共享数据（由它们的源代码来看，的确如此）。

因此，C++11保证了在“独立的内存位置”上，肯定不会发生如上这种问题。或者换种说法：“对于同一内存位置，多个线程同时访问时需要加锁，除非所有的访问都是只读”。需要留意的是，一个单word内的不同比特域并非属于独立的内存位置（译注：即它们总是被一起读取和写入的），所以不要轻易在线程之间共享带有比特域的结构，除非使用锁机制来消除潜在风险。除过这个需要特别注意的地方外，C++的内存模型就如同“大家所期待的那样”，简单而且纯洁）（译注：指对象结构的内存模型就按照结构中的声明顺序进行排列，而且不会发生干扰问题）。

但是，对于底层的并行计算问题，要想直接地进行考虑，并不总是那么简单。考虑下面的代码：

```
// 开始的时候， x==0 , y==0
if (x) y = 1;    // 线程 1
if (y) x = 1;    // 线程 2
```

这里有没有问题呢？或者更直接地说，这里会不会发生资源竞争呢？（答案是：不会发生）

（译注：此处范例的详细解释请翻阅参考资料之“Hans-J. Boehm:Threads basics”）

幸运的是，我们赶上了好时代，每一个现代C++编译器（我所知道的）都给予上面问题正确的答案，而且它们这些年来也是如此做的。毕竟，长期以来C++一直担任着开发并发系统中关键部分的重任，相应的语言标准总不能停滞不前吧。

参考：

- Standard: 1.7 The C++ memory model [intro.memory]

- Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl:

  C++ Data-Dependency Ordering: Atomics and Memory Model.

  N2556==08-0066.

- Hans-J. Boehm:

  Threads basics,

  HPL technical report 2009-259.

  “what every programmer should know about memory model issues.”

- Hans-J. Boehm and Paul McKenney:

  A slightly dated FAQ on C++ memory model issues.

预防窄转换

> （译注："窄转换"是我见到过的一个翻译术语，但我忘记是在那本书上看到的。此处也可译为"预防类型截断"或者"预防类型切割"。）

问题现象：C和C++会进行隐式的（类型）截断

```
int x = 7.3;    // 啊哦!
void f(int);
f(7.3);         // 啊哦!
```

但是，在C++11中，使用{}进行初始化不会发生这种窄转换（译注：也就是使用{}对变量进行初始化时，不会进行隐式的类型截断，编译器会产生一个编译错误，防止隐式的类型截断的发生。）：

```
int x0 {7.3};    // 编译错误: 窄转换
int x1 = {7.3};    // 编译错误：窄转换
double d = 7;
int x2{d};    // 编译错误：窄转换（double类型转化为int类型）
char x3{7};  // OK：虽然7是一个int类型，但这不是窄转换
vector vi = {1, 2.3, 4, 5.6};  //错误：double至int到窄转换
```

C++11避免许多不兼容性的方法是在进行窄转换时尽可能地依赖于用于初始化的实际值（如上例中的7），而非仅仅依赖于变量类型作判断。如果一个值可以无损地用目标类型来存放，那么就不存在窄转换。

```
// OK: 7是一个int类型的数据，但是它可以被无损地表达为char类型数据
char c1{7};
// error: 发生了窄转换，初始值超出了char类型的范围
char c2{77777};
```

请注意，double至int类型的转换通常都会被认为是窄转换，即使从7.0转换至7。

> （评注："{}初始化"对于类型转换的处理增强了C++静态类型系统的安全性。传统的C/C++中依赖于编程人员的初始化类型安全检查在C++11中通过"{}初始化"由编译器实施。）

参考：

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis:

  Initialization and initializers

  (an overview of initialization-related problems with suggested solutions).

- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis:

  Initializer lists.

- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis :

  Initializer lists (Rev. 3) .

- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde:

  Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

nullptr——空指针标识

空指针标识(nullptr)（其本质是一个内定的常量）是一个表示空指针的标识，它不是一个整数。（译注：这里应该与我们常用的NULL宏相区别，虽然它们都是用来表示空置针，但NULL只是一个定义为常整数0的宏，而nullptr是C++0x的一个关键字，一个内建的标识符。下面我们还将看到nullptr与NULL之间更多的区别。）

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0;           //这里0的赋值还是有效的，并且p=p2

void f(int);
void f(char*);

f(0);         //调用f(int)
f(nullptr);   //调用f(char*)

void g(int);
g(nullptr);       //错误：nullptr并不是一个整型常量
int i = nullptr;  //错误：nullptr并不是一个整型常量
```

（译注：实际上，我们这里可以看到nullptr和NULL两者本质的差别，NULL是一个整型数0，而nullptr可以看成是一个char *。）

参考：

- the C++ draft section ???
- [N1488==/03-0071] Herb Sutter and Bjarne Stroustrup:A name for the null pointer: nullptr .
- [N2214 = 07-0074] Herb Sutter and Bjarne Stroustrup:A name for the null pointer: nullptr (revision 4) .

（翻译：张潇）

对重载(override)的控制: override

对重载(override)的控制：final

POD

所谓POD(Plain Old Data)，指的是那些可以像C结构体一样直接操作的“普通”类型，对于该种类型，可以直接对它用memset()/memcpy()来进行初始化/拷贝等操作。

在C++98标准中，POD实际上是受限于结构体定义时所涉之语言特性而定义的。

```
struct S { int a; };    // S属于POD
struct SS { int a; SS(int aa) : a(aa) { } }; // SS不属于POD
struct SSS { virtual void f(); /* ... */ };
```

在C++11中，S和SS都是“标准布局类型”(即POD)，因为SS实在没什么复杂的地方：构造函数不会影响它内存布局（所以memcpy()也能用），不过这里却不能用memset()来初始化——因为它可能违反构造函数中定义的赋值规则(需要用aa来为a赋值）。另外，这里的SSS则明显不是POD了，因为其每个对象中都包含着虚表指针(vptr)。

C++11中引进或重新定义了POD、trivially-copyable类型、trivial类型、以及”标准布局”类型等概念，以用来处理C++98中原”POD”相关的一系列技术问题。

（译注：请参阅《怎样理解C++ 11中的trivial和standard-layout—An answer from stackoverflow》）

POD的(递归)定义如下：

- 所有的成员类型和基类都是POD类型
- 其余部分跟以前一样(参见[10]第9章节)

不含虚函数

不含虚基类

不含引用

不含多种访问权限(译注：对所有non-static成员有相同的public/private/protected访问控制权限)

C++11中关于POD方面最重要的部分就是POD中允许存在不影响内存布局和性能的构造函数（译注：参见C++11中新引入的default构造函数语法）。

参考文献：

- the C++ draft section 3.9 and 9 [10]
- [N2294=07-0154] Beman Dawes:

  POD’s Revisited; Resolving Core Issue 568 (Revision 4)

.

（翻译：张潇）

原生字符串标识

比如，你用标准regex库来写一个正则表达式，但正则表达式中的反斜杠'\'其实却是一个"转义(escape)"操作符(用于特殊字符)，这相当令人讨厌。考虑如何去写"由反斜杠隔开的两个词语"这样一个模式(\w\\w):

```
string s = "\\w\\\\\\w";  // 希望它是对的（译注：不直观、不美观，且容易出错）
```

请注意，在正则表达式和普通C++字符串中，各自都需要使用连续两个反斜杠来表示反斜杠本身。然而，假如使用C++11的原生字符串，反斜杠本身仅需一个反斜杠就可以表示。因而，上述的例子简化为：

```
string s = R"(\w\\\w)";  // 这次百分百正确
```

引发原生字符串标识提议的是这样一个"惊天地泣鬼神"的例子：

```
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"  // 这五个反斜杠是否正确?
       // 即使是专家，也很容易被这么多反斜杠搞得晕头转向
```

R"(…)"记法相比于"…"会有一点点的冗长，但为了不必使用烦琐的"转义(escape)"符号，"多一点"是必要的。

那么，如何将双引号**"**'本身放到原生字符串里呢？只要它不是正好跟在右括弧**)**'之后，那么非常简单：

```
R"("quoted string")"   // 这个字符串是 "quoted string"
```

但是，假如我们偏要在原生字符串中表达右括弧后跟双引号 **)"** 这样一个奇葩组合呢？首先，幸运地是，这种情况一般很少碰到；其次，"(…)"分隔法只不过是默认的分隔语法罢了。通过在**"(…)"**的**(…)**前后添加显式的自定义分隔号(译注:例如下面例子中的三个星号*)，我们还可以创造出任何我们想要的分隔语法。

```
// 字符串为："quoted string containing the usual terminator (")"
R"***("quoted string containing the usual terminator (")")***"
```

在右括弧之后的字符序列(即:自定义分隔号)必须与左括弧之前的字符序列相同。通过这种方式，我们几乎可以处理任意复杂的模式。

参考：

- Standard 2.13.4

- [N2053=06-0123] Beman Dawes: Raw string literals . (original proposal)
- [N2442=07-0312] Lawrence Crowl and Beman Dawes: Raw and Unicode String Literals; Unified Proposal (Rev. 2) . (final proposal combined with the User-defined literals proposal).

（翻译：张潇，dabaitu）

右角括号

考虑如下代码：

```
list<vector<string>> lvs;
```

在C++98中，这是一个语法错误，因为两个右角括号(‘ `>` ’)之间没有空格（译注：因此，编译器会将它分析为” `>>` ”操作符）。C++0x可以正确地分辨出这是两个右角括号(‘ `>` ’)，是两个模板参数列表的结尾。

为什么之前这会是一个问题呢？一般地，一个编译器前端会按照“分析/阶段”模型进行组织。简要描述如下：

- 词法分析（从字符中构造token）
- 语法分析（检查语法）
- 类型检查（确定名称和表达式的类型）

这些阶段在理论上，甚至在某些实际应用中，都是严格独立的。所以，词法分析器会认为” `>>` ;”是一个完整的token（通常意味着右移操作符或是输入），而无法理解它的实际意义（译注：即在具体的上下文环境下，某一个符号的具体意义）。特别地，它无法理解模板或内置模板参数列表。然而，为了使上述示例“正确”，这三个阶段必须进行某种形式的交互、配合。解决这个问题的最关键的点在于，每一个C++ 编译器已完整理解整个问题（译注：对整个问题进行了全部的词法分析、符号分析及类型检测，然后分析各个阶段的正确性），从而给出令人满意的错误消息。

参考：

- the C++ draft section ???
- [N1757==05-0017] Daveed Vandevoorde: revised right angle brackets proposal (revision 2) .

（翻译：张潇，dabaitu）

右值引用

左值（赋值操作符“=”的左侧，通常是一个变量）与右值（赋值操作符“=”的右侧，通常是一个常数、表达式、函数调用）之间的差别可以追溯到 Christopher Strachey （C++的祖先语言CPL之父，指称语义学之父）时代。在C++中，左值可被绑定到非const引用，左值或者右值则可被绑定到const引用。但是却没有什么可以绑定到非const的右值（译注：即右值无法被非const的引用绑定），这是为了防止人们修改临时变量的值，这些临时变量在被赋予新的值之前，都会被销毁。例如：

```
void incr(int& a) { ++a; }
int i = 0;
incr(i);    // i变为1
//错误：0不是一个左值
incr(0);
//（译注：0不是左值，无法直接绑定到非const引用：int&。
// 假如可行，那么在调用时，将会产生一个值为0的临时变量，
// 用于绑定到int&中，但这个临时变量将在函数返回时被销毁，
// 因而，对于它的任何更改都是没有意义的，
// 所以编译器拒绝将临时变量绑定到非const引用，但对于const的引用，
// 则是可行的）
```

假设incr(0)合法，那么要么产生一个程序员不可见的临时变量并进行无意义的递增操作，要么发生更悲剧的情况：把字面常量“0”的实际值会变成1。尽管后者听起来是天方夜谭，但是对于早期Frotran等这一类把字面常量“0”也存到内存里的某个位置的编译器来说，这真的会变成一个bug。（译注：指的是假如把用于存储字面常量0的内存单元上的值从0修改为1以后，后续所有使用字面常数0的地方实际上都在使用“1”）。

到目前为止，一切都很美好。但考虑如下函数：

```
template<class T> swap(T& a, T& b) // 老式的swap函数
{
    T tmp(a);// 现在有两份"a"
    a = b;        // 现在有两份"b"
    b = tmp;    // 现在有两份tmp(值同a)
}
```

如果T是一个拷贝代价相当高昂的类型，例如string和vector，那么上述swap()操作也将煞费气力（不过对于标准库来说，我们已经针对string和vector的swap()进行了特化来处理这个问题）。注意这个奇怪的现象，我们的初衷其实并不是为了把这些变量拷来拷去，我是仅仅是想将变量a,b,tmp的值做一个“移动”（译注：即通过tmp来交换a,b的值）。

在C++11中，我们可以定义“移动构造函数(move constructors)”和“移动赋值操作符(move assignments”来“移动”而非复制它们的参数：

```
template<class T> class vector {
        // …
        vector(const vector&);  // 拷贝构造函数
        vector(vector&&);   // 移动构造函数
        vector& operator= (const vector&); // 拷贝赋值函数
        vector& operator =(vector&&);  // 移动赋值函数
}; //注意：移动构造函数和移动赋值操作符接受
// 非const的右值引用参数，而且通常会对传入的右值引用参数作修改
```

”&&”表示“右值引用”。右值引用可以绑定到右值（但不能绑定到左值）：

```
X a;
X f();
X& r1 = a;        // 将r1绑定到a(一个左值)
X& r2 = f();    // 错误：f()的返回值是右值，无法绑定
X&& rr1 = f();    // OK：将rr1绑定到临时变量
X&& rr2 = a;    // 错误：不能将右值引用rr2绑定到左值a
```

移动赋值操作背后的思想是，“赋值”不一定要通过“拷贝”来做，还可以通过把源对象简单地“偷换”给目标对象来实现。例如对于表达式s1=s2，我们可以不从s2逐字拷贝，而是直接让s1“侵占”s2内部的数据存储(译注：比如char* p)，并以某种方式“删除”s1中原有的数据存储（或者干脆把它扔给s2，因为大多情况下s2随后就会被析构）。（译注：仔细体会copy与move的区别。）

我们如何知道源对象能否“移动”呢？我们可以这样告诉编译器：（译注：通过move()操作符）

```
template <class T>
void swap(T& a, T& b)  //“完美swap”（大多数情况下）
{
      T tmp = move(a);  // 变量a现在失效（译注：内部数据被move到tmp中了）
      a = move(b);      // 变量b现在失效（译注：内部数据被move到a中了，变量a现在“满血复活”了）
      b = move(tmp);    // 变量tmp现在失效（译注：内部数据被move到b中了，变量b现在“满血复活”了）
}
```

move(x) 意味着“你可以把x当做一个右值”，把move()改名为rval()或许会更好，但是事到如今，move()已经使用很多年了。在C++11中，move()模板函数（参考“brief introduction”），以及右值引用被正式引入。

右值引用同时也可以用作完美转发(perfect forwarding)。（译注：比如某个接口函数什么也不做，只是将工作“委派”给其他工作函数）

在C++11的标准库中，所有的容器都提供了移动构造函数和移动赋值操作符，那些插入新元素的操作，如insert()和push_back(), 也都有了可以接受右值引用的版本。最终的结果是，在没有用户干预的情况下，标准容器和算法的性能都提升了，而这些都应归功于拷贝操作的减少。

参考：

- the C++ draft section ???
- N1385 N1690 N1770 N1855 N1952

- [N2027==06-0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki:

  A brief introduction to rvalue references

- [N1377=02-0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams:

  A Proposal to Add Move Semantics Support to the C++ Language (original proposal).

- [N2118=06-0188] Howard Hinnant:

  A Proposal to Add an Rvalue Reference to the C++ Language Proposed Wording (Revision 3) (final proposal).

（翻译：dabaitu，感谢：dave）

Simple SFINAE rule

Stroustrup先生尚未完成这个主题，请稍后再来。

静态（编译期）断言 — static_assert

静态（编译期）断言由一个常量表达式及一个字符串文本构成：

```
static_assert(expression, string);
```

expression在编译期进行求值，当结果为false（即：断言失败）时，将**string**作为错误消息输出。例如：

```
static_assert(sizeof(long) >= 8,
   “64-bit code generation required for this library.”);
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y),
    ”unexpected padding in S”);
```

static_assert在判断代码的编译环境方面（译注：比如判断当前编译环境是否64位）十分有用。但需要注意的是，由于static_assert在编译期进行求值，它不能对那些依赖于运行期计算的值的进行检验。例如：

```
int f(int* p, int n)
{
      //错误：表达式“p == 0”不是一个常量表达式
      static_assert(p == 0,
          “p is not null”);
}
```

（正确的做法是在运行期进行判断，假如条件不成立则抛出异常）

参考：

- the C++ draft 7 [4].
- [N1381==02-0039] Robert Klarer and John Maddock: Proposal to Add Static Assertions to the Core Language .
- [N1720==04-0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant: Proposal to Add Static Assertions to the Core Language (Revision 3) .

（翻译：张潇，dabaitu）

模板别名（正式的名称为"template typedef"）

如何在某个模板类的基础上，通过写死（绑定）部分模板参数类型，来定制出一个新的模板类？

尝尝我这种写法如何：

```
template<class T>
// 采用了自定义内存分配器的std::vector
using Vec = std::vector<T,My_alloc<T>>;

// 使用My_alloc为元素分配存储空间
Vec<int> fib = { 1, 2, 3, 5, 8, 13 };

// verbose与fib类型一致
vector<int,My_alloc<int>> verbose = fib;
```

有了using语法，定义模板别名则可一目了然：***using*** 模板别名 **=** 引用细节**;**。在这之前，我们也曾徘徊在"typedef"的经典和复杂之间左右为难，但没有哪一个方案能做到完美平衡，直到后来我们干脆弃而转向言简意赅的using语法。

即使模板存在特化，using语法也能照常使用。（需注意：为模板及其各种特化形式可以设定一个统一的别名，但反之则不然：模板特化操作不能通过别名进行）。例如：

```
template<int>
// idea: int_exact_trait<N>::type用于表达含有N个bit的数值类型
struct int_exact_traits {
    typedef int type;
};

template<>
struct int_exact_traits<8> {
    typedef char type;
};

template<>
struct int_exact_traits<16> {
    typedef char[2] type;
};

// ...

template<int N>
// 定义别名用以简化书写
// 译注：给模板的通用版本取别名，则其所有的特化版本自动获得该别名，
//     例如对于8bit的特化版本，现在可直接使用别名
    using int_exact = typename int_exact_traits<N>::type;
    // int_exact<8> 是含有8个bit的数值类型
int_exact<8> a = 7;
```

除了在模板方面身担重任外，using语法也可作为对普通类型定义别名的另一种选择（相较于typedef更合吾意）。

```
typedef void (*PFD)(double);    // C 样式
using PF = void (*)(double);    // using加上C样式的类型
using P = [](double)->void;  // using和函数返回类型后置语法
```

参考：

参考：

- the C++ draft: 14.6.7 Template aliases; 7.1.3 The typedef specifier
- [N1489=03-0072] Bjarne Stroustrup and Gabriel Dos Reis: Templates aliases for C++ .
- [N2258=07-0118] Gabriel Dos Reis and Bjarne Stroustrup: Templates Aliases (Revision 3) (final proposal).

（翻译：张潇，dabaitu）

线程本地化存储 (thread_local)

抱歉，目前我还没有完成这个主题，请稍后再来。

如果你对这一主题感兴趣，可以参考本站的：

- C++小品：井水不犯河水的thread_specific_ptr，C++11线程库中的本地存储
- C++小品：井水不犯河水在PPL中的实现：combinable以及task_group,task

参考：

- [N2659 = 08-0169] Lawrence Crowl:

  Thread-Local Storage (Final proposal).

unicode字符

抱歉，目前我尚未完成这个主题，请稍后再来。

（译注：C++对unicode的支持不是特别重视）

统一初始化的语法和语义

按照对象的类型以及初始化时的上下文，C++提供了五花八门的对象初始化的方式。若不慎误用，可能会产生匪夷所思的谬误，而且还伴随着莫名其妙的错误（调试）信息。考虑如下的代码：

```
string a[] = { “foo”, ” bar” };  //正确：初始化数组变量
//错误：初始化列表应用在了非聚合的向量上
vector<string> v = { “foo”, ” bar” };
void f(string a[]);
f( { “foo”, ” bar” } );   //语法错误，把一个块（block）作为了参数
```

以及：

```
int a = 2;         //“赋值风格”的初始化
int aa[] = { 2, 3 }; //用初始化列表进行的赋值风格的初始化
complex z(1,2);   //“函数风格”的初始化
x = Ptr(y);     // “函数风格”的转换/赋值/构造操作
```

再如：

```
int a(1);  //变量的定义
int b();    //函数的声明
int b(foo);    // 变量的定义，或者函数的声明
```

要记住这么多种初始化规则，并从中选用最合适的一种，绝非易事。

C++11的解决方法是对于所有的初始化，均可使用“{}-初始化变量列表”：

```
X x1 = X{1,2};
X x2 = {1,2};     // 此处的'='可有可无
X x3{1,2};
X* p = new X{1,2};

struct D : X {
    D(int x, int y) :X{x,y} { /* … */ };
};

struct S {
    int a[3];
    // 对于旧有问题的解决方案
    S(int x, int y, int z) :a{x,y,z} { /* … */ };
};
```

与以往相比最为关键的变动是，X{a}方式的初始化，在所有的语境中都能构造出同样的结果，所以凡是能用“{}”的初始化，得到的结果都是一致的。例如：

```
X x{a};
X* p = new X{a};
z = X{a};          // 使用了类型转换
f({a});            // a作为函数的X型实参
return {a};        // a作为函数的X型返回值
```

其他参考文档：

- the C++ draft section ???
- [N2215==07-0075] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists (Rev. 3) .
- [N2640==08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists — Alternative Mechanism and Rationale (v. 2) (final proposal).

（翻译：张潇）

（广义的）联合体

在C++98（或更早）版本中，union的成员类型，必须不含自定义构造/析构函数或者赋值操作符。

```
union U {
  int m1;
  complex m2;    //错误（明显的）：complex拥有构造函数
  //错误（不那么明显）：string的内部数据只能严格地由其构造函数，
  // 拷贝构造函数，和析构函数去维护
  string m3;
};
```

亦即：

```
U u;        // 使用哪个成员的构造函数呢？
u.m1 = 1;     // 给整型成员赋值
string s = u.m3;    //灾难：从string成员拷贝
```

显而易见，把值写入一个成员，之后又读取另外一个成员的做法是给自己找麻烦，然而人们往往并非刻意而为（多因失误导致）。

C++11对union的限制条件进行了放宽，以允许更灵活广泛的成员类型。其中值得特别指出的是，带有自定义构造函数/析构函数的类型现在也可作为union的成员了。此外，为使“灵活”不至成为脱缰野马，新标准又特别引入了“第四条军规”（译注：参见下文），并倡导使用“可识别union”（译注：参见下文Widget样例以及Boost::Any和Boost::Variant）。

C++11中的对union的限制条件重新定义如下：

- 不含虚函数（与C++98相同）
- 不含引用成员（与C++98相同）
- 没有基类（与C++98相同）
- 若union的某个成员的类型含有自定义构造/拷贝/析构函数，那么该union的相应构造/拷贝/析构函数将会被自动“禁用”（译注：在C++11中我们可以使用delete关键字来“禁用”构造/析构函数），随之而来的后果是：该union不能被实例化成对象。（新增的所谓“第四条军规”）

例如：

```
union U1 {
  int m1;
  complex m2;    // ok
};

union U2 {
  int m1;
  string m3;    // ok
};
```

上述代码看起来容易出问题（译注：例如有人实例化了U2类型的对象并给其m1成员赋值之后，当union析构时，m3的析构函数可能会crash），但是有了第四条军规，刚才的隐含问题便迎刃而解（译注：通过编译错误）。即：

```
U1 u;          // ok
u.m2 = {1,2};    // ok：给complex成员赋值
U2 u2;   // 编译错误: string类含有析构函数，因而U2的析构函数已被自动禁用
    //（译注：析构函数被禁用意味着不允许在栈上实例化U2对象，否则无法析构）
U2 u3 = u2;    // 编译错误：string类含有拷贝构造函数，
    // 因而U2类型同样也不能被拷贝构造
```

这样看来，先前定义的U2几乎没什么实际用途了。唯一可能用到这种奇葩union的地方是，把它嵌到结构体里并且额外记录其“当值”成员类型——也就是所谓的“可识别union”。样例如下：

```
class Widget {  // 用union存储的“三态”Widget
  private:
    // 用以实现“可识别union”的“当值”类型标记
    enum class Tag { point, number, text } type;
    union {    // 三态的具体存储形式
      point p;    // point类含有构造函数
      int i;
      string s;   // string类含有默认的构造/拷贝/析构函数
    };
    // ...
    // 由于string中存在拷贝函数，所以需要“手工拷贝”union
    Widget& operator=(const Widget& w)
    {
      // 译注：从text态到text态
      if (type==Tag::text && w.type==Tag::text) {
        s = w.s;    // 直接使用string的赋值运算符
        return *this;
      }
      // 译注：从text态到其他态，意味着union不再用于存放string
      //    由于union的拷贝函数已被自动禁用，
      //    所以需要有人手工释放string原先所占资源
      if (type==Tag::text) s.~string();  // 此处需要显式析构

      switch (w.type) {
      case Tag::point: p = w.p; break;  // 普通的拷贝
      case Tag::number: i = w.i; break;
      // 译注：C++98/03标准中的的原地拷贝构造语法
      case Tag::text: new(&s)(w.s); break;  // placement new
      }
      type = w.type;
      return *this;
    }
  };
};
```

其他参考文献：

- [N2544=08-0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer: Unrestricted unions (Revison 2)

（翻译：张潇）

用户定义数据标识（User-defined literals）

C++提供了许多内建数据类型的数据标识（2.14节变量）：

```
123    // int整型
1.2    // double双精度型
1.2F    // float浮点型
'a'    // char字符型
1ULL    // unsigned long long64位无符号长整型
0xD0    // hexadecimal unsigned十六进制无符号整型
"as"    // string字符串
```

但是在C++98中并没有为用户自定义的变量类型提供数据标识。这就违反甚至冲突于“用户自定类型应该和内建类型一样得到支持”的原则。在特殊情况下，人们有以下的需求：

```
"Hi!"s            //字符串，不是“以零字符为终结的字符数组”
1.2i            //虚数
123.4567891234df    //十进制浮点型（IBM）
101010111000101b    //二进制
123s            //秒
123.56km            //不是英里（单位）
1234567890123456789012345678901234567890x        //扩展精度
```

C++11通过在变量后面加上一个后缀来标定所需的类型以支持“用户定义数据标识”，例如：

```
constexpr complex operator "" i(long double d)    // 设计中的数据标识
{
    return {0,d};    //complex是一个数据标识
}

// 将n个字符构造成字符串std::string对象的数据标识
std::string operator""s (const char* p, size_t n)
{
    return string(p,n);    // 需要释放存储空间
}
```

这里需要注意的是，constexpr的使用可以进行编译时期的计算。使用这一功能，我们可以这样写：

```
template <class T> void f(const T&);
f("Hello");    // 传递char*指针给f()
f("Hello"s);    // 传递（5个字符的）字符串对象给f()
f("Hello n"s);    // 传递（6个字符的）字符串对象给f()

auto z = 2+1i;    // 复数complex(2,1)
```

基本（实现）方法是编译器在解析什么语句代表一个变量之后，再分析一下后缀。用户自定义数据标识机制只是简简单单的允许用户制定一个新的后缀，并决定如何对它之前的数据进行处理。要想重新定义一个内建的数据标识的意义或者它的参数、语法是不可能的。一个数

据标识操作符可以使用它（前面）的数据标识传递过来的处理过的值（如果是使用新的没有定义过的后缀的值）或者没有处理过的值（作为一个字符串）。

要得到一个没有处理过的字符串，只要使用一个单独的const char*参数即可，例如：

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);
f(1234567890123456789012345678901234567890x);
```

这个C语言风格的字符串”1234567890123456789012345678901234567890″被传递给了操作符 operator”” x()。注意，我们并没有明确地把数字转换成字符串。

有以下四种数据标识的情况，可以被用户定义后缀来使用用户自定义数据标识：

- 整型标识：允许传入一个unsigned long long或者const char*参数
- 浮点型标识：允许传入一个long double或者const char*参数
- 字符串标识：允许传入一组(const char*,size_t)参数
- 字符标识：允许传入一个char参数。

注意，你为字符串标识定义的标识操作符不能只带有一个const char*参数（而没有大小）。例如：

```
//警告，这个标识操作符并不能像预想的那样子工作
string operator"" S(const char* p);
"one two"S;    //错误，没有适用的标识操作符
```

根本原因是如果我们想有一个“不同的字符串”，我们同时也想知道字符的个数。后缀可能比较短（例如，s是字符串的后缀，i是虚数的后缀，m是米的后缀，x是扩展类型的后缀），所以不同的用法很容易产生冲突，我们可以使用namespace（命名空间）来避免这些名字冲突：

```
namespace Numerics {
    // …
    class Bignum { /* … */ };
    namespace literals {
        operator"" X(char const*);
    }
}

using namespace Numerics::literals;
```

参考：

- Standard 2.14.8 User-defined literals
- [N2378==07-0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde:

User-defined Literals (aka. Extensible Literals (revision 3)).

可变参数模板（Variadic Templates）

要解决的问题：

- 怎么创建一个拥有1个、2个或者更多的初始化器的类？
- 怎么避免创建一个实例而只拷贝部分的结果？
- 怎么创建一个元组？

最后的问题是关键所在：考虑一下元组！如果你能创建并且访问一般的元组，那么剩下的问题也将迎刃而解。

这里有一个例子（摘自“可变参数模板简述（A brief introduction to Variadic templates）”（参见参考）），要构建一个广义的、类型安全的printf()。这个方法比用boost::format好的多，但是考虑一下：

```
const string pi = “pi”;
const char* m =
    “The value of %s is about %g (unless you live in %s).n”;
printf(m,  pi, 3.14159,  “Indiana”);
```

这是除了格式字符串之外，没有其它参数的情况下调用printf()的一个最简单的例子了，所以我们将要首先解决：

```
void printf(const char* s)
{
 while (s && *s) {
      if (*s==’%’ && *++s!=’%') //保证没有更多的参数了
//%%（转义字符，在格式字符串中代表%
          throw runtime_error(“格式非法：缺少参数”);
     std::cout << *s++<<endl;
 }
}
```

这个处理好之后，我们必须处理有更多参数的printf()：

```
template<typename T, typename... Args>    // 注意这里的"..."
void printf(const char* s, T value, Args... args)   // 注意"..."
{
    while (s && *s) {
        //一个格式标记（避免格式控制符）
                        if (*s=='%' && *++s!='%') {
            std::cout << value;
            return printf(++s, args...);//使用第一个非格式参数
        }
        std::cout << *s++;
    }
    throw std::runtime error("extra args provided to printf");
}
```

这段代码简单地“去除”了开头的无格式参数，之后递归地调用自己。当没有更多的无格式参数的时候，它调用第一个（很简单）printf()（如上所示）。这也是标准的函数式编程在编译的时候做的(?)。注意，`<<` 的重载代替了在格式控制符当中（可能会有错误）的花哨的技巧。（译注：我想这里可能指的是使用重载的 `<<` 输出操作符，就可以避免使用各种技巧复杂的格式控制字符串。） Args…定义的是一个叫做“参数包”的东西。这个“参数包”仅仅是一个（有各种类型的值的）队列，而且这个队列中的参数可以从头开始进行剥离（处理）。如果我们使用一个参数调用printf()，函数的第一个定义(printf(const char))就被调用。如果我们使用两个或者更多的参数调用*printf()*，那么函数的第二个定义*(printf(const char*, T value, Args… args))就会被调用，把第一个参数当作字符串，第二个参数当作值，而剩余的参数都打包到参数包args中，用做函数内部的使用。在下面的调用中：

```
printf(++s, args…);
```

参数包args被打开，所以参数包中的下一个参数被选择作为值。这个过程会持续进行，直到args为空（所以第一个printf()最终会被调用）。

如果你对函数式编程很熟悉的话，你可能会发现这个语法和标准技术有一点不一样。如果发现了，这里有一些小的技术示例可能会帮助你理解。首先我们可以声明一个普通的可变参数函数模板（就像上面的printf()）：

```
template<class ... Types>
// 可变参数模板函数
//（补充：一个函数可以接受若干个类型的若干个参数）
void f(Types ... args);

f();        // OK: args不包含任何参数
f(1);       // OK: args有一个参数: int
f(2, 1.0);  // OK: args有两个参数: int和double
```

我们可以建立一个具有可变参数的元组类型：

```
template<typename Head, typename... Tail>
//这里是一个递归
//一个元组最基本要存储它的head（第一个（类型/值））对
//并且派生自它的tail（剩余的（类型/值））对
//注意，这里的类型被编码，而不是按一个数据来存储
class tuple<Head, Tail...>
    : private tuple<Tail...> {
    typedef tuple<Tail...> inherited;
public:
    tuple() { } // 默认的空tuple

    //从分离的参数中创建元组
    tuple(typename add_const_reference<Head>::type v,
 typename add_const_reference<Tail>::type... vtail)
        : m_head(v), inherited(vtail...) { }

    // 从另外一个tuple创建tuple:
    template<typename... VValues>
    tuple(const tuple<VValues...>& other)
    :    m_head(other.head()), inherited(other.tail()) { }

    template<typename... VValues>
    tuple& operator=(const tuple<VValues...>& other)  // 等于操作
    {
        m_head = other.head();
        tail() = other.tail();
        return *this;
    }

    typename add_reference<Head>::type head()
            { return m_head; }
    typename add_reference<const Head>::type head() const
            { return m_head; }

    inherited& tail() { return *this; }
    const inherited& tail() const { return *this; }
protected:
    Head m_head;
}
```

有了定义之后，我们可以创建元组（并且复制和操作它们）：

```
tuple<string,vector,double> tt("hello",{1,2,3,4},1.2);
string h = tt.head();   // "hello"
tuple<vector<int>,double> t2 = tt.tail();
```

要实现所有的数据类型可能会比较乏味，所以我们经常减少参数的类型，例如，可以使用标准库中的make_tuple()函数：

```
template<class... Types>
// 这个定义十分简单（参见标准20.5.2.2）
tuple<Types...> make_tuple(Types&&... t)
{
    return tuple<Types...>(t...);
}

string s = "Hello";
vector<int> v = {1,22,3,4,5};
auto x = make_tuple(s,v,1.2);
```

参考：

- Standard 14.6.3 Variadic templates
- [N2151==07-0011] D. Gregor, J. Jarvi:

  Variadic Templates for the C++0x Standard Library.

- [N2080==06-0150] D. Gregor, J. Jarvi, G. Powell:

  Variadic Templates (Revision 3).

- [N2087==06-0157] Douglas Gregor:

  A Brief Introduction to Variadic Templates.

- [N2772==08-0282] L. Joly, R. Klarer:

  Variadic functions: Variadic templates or initializer lists? — Revision 1.

- [N2551==08-0061] Sylvain Pion:

  A variadic std::min(T, ...) for the C++ Standard Library (Revision 2) .

- Anthony Williams:

  An introduction to Variadic Templates in C++0x.

  DevX.com, May 2009.

关于标准库的问题

abandoning_a_process

Stroustrup尚未完成此主题，期待中。

参考：

- Abandoning a process

（翻译：interma）

算法方面的改进

标准库的算法部分进行了如下改进：新增了一些算法函数；通过新语言特性改善了一些算法实现并且更易于使用。下面分别来看一些例子：

- 新算法:

```
bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

Iter find_if_not(Iter first, Iter last, Pred pred);

OutIter copy_if(InIter first, InIter last,
        OutIter result, Pred pred);
OutIter copy_n(InIter first, InIter::difference_type n,
        OutIter result);

OutIter move(InIter first, InIter last, OutIter result);
OutIter move_backward(InIter first, InIter last, OutIter result);

pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last,
        OutIter1 out_true, OutIter2 out_false, Pred pred);
Iter partition_point(Iter first, Iter last, Pred pred);

RAIter partial_sort_copy(InIter first, InIter last,
        RAIter result_first, RAIter result_last);
RAIter partial_sort_copy(InIter first, InIter last,
        RAIter result_first, RAIter result_last, Compare comp);
bool is_sorted(Iter first, Iter last);
bool is_sorted(Iter first, Iter last, Compare comp);
Iter is_sorted_until(Iter first, Iter last);
Iter is_sorted_until(Iter first, Iter last, Compare comp);

bool is_heap(Iter first, Iter last);
bool is_heap(Iter first, Iter last, Compare comp);
Iter is_heap_until(Iter first, Iter last);
Iter is_heap_until(Iter first, Iter last, Compare comp);

T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a,
        const T& b,
         Compare comp);
pair<const T&, const T&> minmax(initializer_list<T> t);
pair<const T&, const T&> minmax(initializer_list<T> t,
         Compare comp);
pair<Iter, Iter> minmax_element(Iter first, Iter last);
pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

// 填充[first,last]范围内的每一个元素
// 第一个元素为value，第二个为++value，以此类ui
// 等同于
// *(d_first)   = value;
// *(d_first+1) = ++value;
// *(d_first+2) = ++value;
// *(d_first+3) = ++value; ...
// 注意函数名，是iota而不是itoa哦
void iota(Iter first, Iter last, T value);
```

- 更有效的move：more操作比copy操作更有效率（参看move semantics（译注：实际上是一个右值(rval)问题，核心是减少创建不必要的对象））。基于move的std::sort()和std::set::insert()要比基于copy的对应版本快15倍以上。不过它对标准库中已有操作的性能改善不多，因为它们的实现中已经使用了类似的方法进行优化了（例如string，vector使用了调优过的swap操作来代替copy了）。当然如果你自己的代码中包含了move操作的话，就能自动从新标准库中获益了。试着用move操作来替代下边这个sort函数中的智能指针（unique_ptr）吧（译注：可以通过一个move版swap来搞定，参看之前move semantics文章）：

```
template<class P> struct Cmp<P> { //比较 *P 的值
    bool operator() (P a, P b) const
                { return *a<*b; }
}

vector<std::unique_ptr<Big>> vb;
//  用指向大对象的unique_ptr填充vb

sort(vb.begin(),vb.end(),Cmp<Big>());// 不要像这样使用时用auto_ptr
```

- 对lambda表达式的使用：对于为标准库算法写函数/函数对象（function object，推荐）这个事儿大家已经抱怨很久了（例如Cmp）。特别是在C++98标准中，这会令人更加痛苦，因为无法定义一个局部的函数对象。不过现在好多了，lambda表达式允许用"inline"的方式来写函数了：

```
sort(vb.begin(),vb.end(),
     [](unique_ptr a, unique_ptr b) { return *a< *b; });
```

  我希望大家尽量多用用lambda表达式（它真的是一种很好很强大的机制）（译注：原文是："I expect lambdas to be a bit overused initially (like all powerful mechanisms)"，非字面翻译）

- 对初始化列表（initializer lists）的使用：初始化表有时可以像参数那样方便的使用。看下边这个例子（x,y,z是string变量，Nocase是一个大小写不敏感的比较函数）：

```
auto x = max({x,y,z},Nocase());
```

参考：

- 25 Algorithms library [algorithms]
- 26.7 Generalized numeric operations [numeric.ops]
- Howard E. Hinnant, Peter Dimov, and Dave Abrahams:

  A Proposal to Add Move Semantics Support to the C++ Language.

  N1377=02-0035.

array

std::array是一个支持随机访问且大小（size）固定的容器（译注：可以认为是一个紧缩版的vector吧）。它有如下特点：

- 不预留多余空间，只分配必须空间（译注：size() == capacity()）。
- 可以使用初始化表（initializer list）的方式进行初始化。
- 保存了自己的size信息。
- 不支持隐式指针类型转换。

换句话说，可以认为它是一个很不错的内建数组类型。一些代码片段：

```
array<int,6> a = { 1, 2, 3 };
a[3]=4;
int x = a[5];    // array的默认数据元素为0，所以x的值变成0
int* p1 = a; // 错误: std::array不能隐式地转换为指针
int* p2 = a.data(); // 正确，data()得到指向第一个元素的指针
```

不过要注意：是可以定义一个长度为0的array的；但是无法从初始化表中推导出size信息：

```
array<int> a3 = { 1, 2, 3 };  //错误：没有size信息
array<int,0> a0;   // 正确: 没有任何元素
int* p = a0.data();  // 为定义行为，不要这样做
```

array非常适合在嵌入式系统（和有类似限制/性能敏感/安全关键系统等）中使用。它提供了序列型容器该有的大部分通用函数（和vector很像）：

```
template<class C> C::value_type sum(const C& a)
{
    return accumulate(a.begin(),a.end(),0);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// …
int x1 = sum(a10);
int x2 = sum(a1000);
int x3 = sum(v);
```

但是，它是不支持由子类到基类的自动类型转换的（注意这个潜在陷阱）：

```
struct Apple : Fruit { /* … */ };
struct Pear : Fruit { /* … */ };

void nasty(array<fruit *,10>& f)
{
        f[7] = new Pear();
};

array<apple ,10> apples;
// …
nasty(apples);  // 错误: 不能将array转换为array;
```

如果支持这种转换的话，apple array中就能放pear啦。

参考：

- Standard: 23.3.1 Class template array

（翻译：interma）

async()

async()函数是一个简单任务的"启动"（launcher）函数，它是本FAQ中唯一一个尚未在标准草案中投票通过的特性。我希望它能在调和两个略微不同的意见之后最终于10月份获得通过（记得随时骚扰你那边的投票委员，一定要为它投票啊?）。

下边是一种优于传统的线程+锁的并发编程方法示例(译注：山寨map-reduce哦)：

```
template<class T,class V> struct Accum  { // 简单的积函数对象
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee}, val{vv} {}
    V operator() ()
    { return std::accumulate(b,e,val); }
};

void comp(vector<double>& v)
    // 如果v够大，则产生很多任务        {
    if (v.size()<10000)
        return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

尽管这只是一个简单的并发编程示例（留意其中的"magic number"），不过我们可没有使用线程，锁，缓冲区等概念。f*变量的类型（即async()的返回值）是"std::future"类型。future.get()表示如果有必要的话则等待相应的线程(std::thread)运行结束。async的工作是根据需要来启动新线程，而future的工作则是等待新线程运行结束。"简单性"是async/future设计中最重视的一个方面；future一般也可以和线程一起使用，不过不要使用async()来启动类似I/O操作，操作互斥体（mutex），多任务交互操作等复杂任务。async()背后的理念和range-for statement很类似：简单事儿简单做，把复杂的事情留给一般的通用机制来搞定吧。

async()可以启动一个新线程或者复用一个它认为合适的已有线程（非调用线程即可）(译注：语义上并发即可，不关心具体的调度策略。和go语义中的goroutines有点像)。后者从用户视角看更有效一些（只对简单任务而言）。

参考：

- Standard: ???
- Lawrence Crowl:

  An Asynchronous Call for C++.

  N2889 = 09-0079.

- Herb Sutter :

  A simple async()

  N2901 = 09-0091 .

（翻译：interma）

atomic_operations

条件变量（Condition variables）

条件变量（Condition variables）提供了同步语义，它会将一个线程阻塞住(block)直到被其他线程所唤醒(notify)或到达了系统超时时间。

不过Stroustrup尚未完成此主题，期待中。

参看：

- Standard: 30.5 Condition variables(thread.condition)

（翻译：interma）

标准库中容器方面的改进

新的语言特性和近10年来的经验会给标准库中的容器带来啥改进呢？首先，新容器类型：array(大小固定容器)，forward_list

（单向链表），unordered containers（哈希表，无序容器）。其次，新特性：initializer lists（初始化列表），rvalue

references（右值引用），variadic templates（可变参数模板），constexpr（常量表达式）。下边均以vector为例加以介绍：

- 初始化列表（Initializer lists）：最显著的改进是容器的构造函数可以接受初始化列表来作为参数：

```
vector<string> vs = { "Hello", ", ", "World!", "\n" };
for (auto s : vs ) cout << s;
```

- move操作：容器新增了move版的构造和赋值函数（作为传统copy操作的补充）。它最重要的内涵就是允许我们高效的从函数中返回一个容器：

```
vector<int&gt; make_random(int n)
{
    vector<int&gt; ref(n);
    // 产生0-255之间的随机数
        for(auto x& : ref) x = rand_int(0,255);

        return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000)) cout << x << '\n';
```

  上边代码的关键点是vector没有被拷贝操作(译注：vector ref的内存空间不是应该在函数返回时被stack自动回收吗？move assignment通过右值引用精巧的搞定了这个问题)。对比我们现在的两种惯用法：在自由存储区来分配vector的空间，我们得负担上内存管理的问题了；通过参数传进已经分配好空间的vector，我们得要写不太美观的代码了（同时也增加了出错的可能）。

- 改进的push操作：作为我最喜爱的容器操作函数，push_back()允许我们优雅的增大容器：

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin&gt;&gt;s&gt;&gt;i) vp.push_back({s,i});
```

如上代码通过r和i构造了一个pair对象，然后将它move到vp中。注意这里是"move"而不是"copy"。这个push_back版本接受了一个右值引用参数，因此我们可以从string的移动构造函数（move

constructor）（译注：直接由拷贝构造函数（copy ctor）对应而来）中获益。同时使用了[统一初始化语法]（unified initializer syntax）来避免啰嗦。

- 原地安置操作（Emplace operations）：在大多数情况下，push_back()使用移动构造函数（而不是拷贝构造函数）来保证它更有效率，不过在极端情况下我们可以走的更远。为何一定要进行拷贝/移动操作？为什么不能在vector中分配好空间，然后直接在这个空间上构造我们需要的对象呢？做这种事儿的操作被叫做"原地安置"（emplace，含义是：putting in place）。举一个emplace_back()的例子：

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin&gt;&gt;s&gt;&gt;i) vp.emplace_back(s,i);
```

  emplace_back()接受了可变参数模板变量并通过它来构造所需类型。至于emplace_back()是否比push_back()更有效率，取决于它和可变参数模板的具体实现。如果你认为这是一个重要的问题，那就实际测试一下。否则，就从美感上来选择它们吧。到目前为止，我更喜欢push_back()，不过只是目前哦。

- Scoped allocators：现在容器中可以持有"拥有状态的空间分配对象(allocationobjects)"了，并通过它来进行"nested/scoped"方式的空间分配（译注：原文：use those to control nested/scoped allocation ）（举例：为容器中的元素分配空间）。

显然，容器不是唯一从新语言特性中获益的标准库部分：

- 编译期计算（Compile-time evaluation）：常量表达式

  为bitset, duration, char_traits, array, atomic types,

  random numbers,

  complex等类型引入了编译期计算。对于某些情况，这意味着性能上的改善；而对于其它情况（无法编译期优化的情况下），则意味着可以减少晦涩代码和宏的使用了。

- 元组（Tuples）：如果没有可变参数模板

  ，它就不存在了。

（翻译：interma）

std::function 和 std::bind

标准库函数bind()和function()定义于头文件 `<functional>` 中（该头文件还包括许多其他函数对象），用于处理函数及函数参数。bind()接受一个函数（或者函数对象，或者任何你可以通过"(…)"符号调用的事物），生成一个其有某一个或多个函数参数被"绑定"或重新组织的函数对象。（译注：顾名思义，bind()函数的意义就像它的函数名一样，是用来绑定函数调用的某些参数的。）例如：

```
int f(int, char, double);
// 绑定f()函数调用的第二个和第三个参数，
// 返回一个新的函数对象为ff，它只带有一个int类型的参数
auto ff = bind(f, _1, 'c', 1.2);
int x = ff(7);                // f(7, 'c', 1.2);
```

参数的绑定通常称为"Currying"（译注：Currying—"烹制咖喱烧菜"，此处意指对函数或函数对象进行加工修饰操作），"_1″是一个占位符对象，用于表示当函数f通过函数ff进行调用时，函数ff的第一个参数在函数f的参数列表中的位置。第一个参数称为"_1″, 第二个参数为"_2″，依此类推。例如：

```
int f(int, char, double);
auto frev = bind(f, _3, _2, _1);        // 翻转参数顺序
int x = frev(1.2, 'c', 7);            // f(7, 'c', 1.2);
```

此处，auto关键字节约了我们去推断bind返回的结果类型的工作。

我们无法使用bind()绑定一个重载函数的参数，我们必须显式地指出需要绑定的重载函数的版本：

```
int g(int);
double g(double);

auto g1 = bind(g, _1);    // 错误：调用哪一个g()？
// 正确，但是相当丑陋
auto g2 = bind( (double(*)(double))g, _1);
```

bind()有两种版本：一个如上所述，另一个则是"历史遗留"的版本：你可以显式地描述返回类型。例如：

```
auto f2 = bind<int> (f, 7, 'c', _1);      // 显式返回类型
int x = f2(1.2);                    // f(7, 'c', 1.2);
```

第二种形式的存在是必要的，并且因为第一个版本（(?) "and for a user simplest "，此处请参考原文)）无法在C++98中实现。所以第二个版本已经被广泛使用。

function是一个拥有任何可以以"(...)"符号进行调用的值的类型。特别地，bind的返回结果可以赋值给function类型。function十分易于使用。（译注：更直观地，可以把function看成是一种表示函数的数据类型，就像函数对象一样。只不过普通的数据类型表示的是数据，function表示的是函数这个抽象概念。）例如：

```
// 构造一个函数对象，
// 它能表示的是一个返回值为float，
// 两个参数为int，int的函数
function<float (int x, int y)> f;

// 构造一个可以使用"()"进行调用的函数对象类型
struct int_div {
        float operator() (int x, int y) const
             { return ((float)x)/y; };
};

f = int_div();                        // 赋值
cout<< f(5,3) <<endl;  // 通过函数对象进行调用
std::accumulate(b, e, 1, f);        // 完美传递
```

成员函数可被看做是带有额外参数的自由函数：

```
struct X {
        int foo(int);
};

// 所谓的额外参数，
// 就是成员函数默认的第一个参数，
// 也就是指向调用成员函数的对象的this指针
function<int (X*, int)> f;
f = &X::foo;            // 指向成员函数

X x;
int v = f(&x, 5);  // 在对象x上用参数5调用X::foo()
function<int (int)> ff = std::bind(f, &x, _1);    // f的第一个参数是&x
v = ff(5);                // 调用x.foo(5)
```

function对于回调函数、将操作作为参数传递等十分有用。它可以看做是C++98标准库中函数对象mem_fun_t, pointer_to_unary_function等的替代品。同样的，bind()也可以被看做是bind1st()和bind2nd()的替代品，当然比他们更强大更灵活。

参考：

- Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function
- Herb Sutter:Generalized Function Pointers

  .

  August 2003.

- Douglas Gregor:

  Boost.Function

  .

- Boost::bind

（翻译：dabaitu）

std::forward_list

std::forward_list是一个基本的单向链表。它只提供了前向迭代器（forward

iteration）；并在执行插入/删除操作后，其他节点也不会受到影响（译注：其它迭代器不失效）。它尽可能减少所占用空间的大小（空链表很可能只占用一个word2

（译注：2Byte））且不提供size()操作（所以也没有存储size的数据成员），简略原型如下：

```
template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
class forward_list {
public:
    // the usual container stuff
    // no size()
    // no reverse iteration
    // no back() or push_back()
};
```

参看:

- Standard: 23.3.3 Class template forward_list

（翻译：interma）

std::future和std::promise

并行开发挺复杂的，特别是在试图用好线程和锁的过程中。如果要用到条件变量或std-atomics（一种无锁开发方式），那就更复杂了。C++0x提供了future和promise来简化任务线程间的返回值操作；同时为启动任务线程提供了packaged_task以方便操作。其中的关键点是允许2个任务间使用无（显式）锁的方式进行值传递；标准库帮你高效的做好这些了。基本思路很简单：当一个任务需要向父线程（启动它的线程）返回值时，它把这个值放到promise中。之后，这个返回值会出现在和此promise关联的future中。于是父线程就能读到返回值。更简单点的方法，参看async()。

标准库中提供了3种future：普通future和为复杂场合使用的shared_future和atomic_future。在本主题中，只展示了普通future，它已经完全够用了。如果我们有一个future

f，通过get()可以获得它的值：

```
X v = f.get();  // if necessary wait for the value to get computed
```

如果它的返回值还没有到达，调用线程会进行阻塞等待。要是等啊等啊，等到花儿也谢了的话，get()会抛出异常的（从标准库或等待的线程那个线程中抛出）。

如果我们不需要等待返回值（非阻塞方式），可以简单询问一下future，看返回值是否已经到达：

```
if (f.wait_for(0))
{
    // there is a value to get()
    // do something
}
else
{
    // do something else
}
```

但是，future最主要的目的还是提供一个简单的获取返回值的方法：get()。

promise的主要目的是提供一个”put”（或”get”，随你）操作，以和future的get()对应。future和promise的名字是有历史来历的，是一个双关语。感觉有点别扭？请别怪我。

promise为future传递的结果类型有2种：传一个普通值或者抛出一个异常

```
try {
        X res;
        // compute a value for res
        p.set_value(res);
}
catch (…) {   // oops: couldn’t compute res
        p.set_exception(std::current_exception());
}
```

到目前为止还不错，不过我们如何匹配future/promise对呢？一个在我的线程，另一个在别的啥线程中吗？是这样：既然future和promise可以被到处移动（不是拷贝），那么可能性就挺多的。最普遍的情况是父子线程配对形式，父线程用future获取子线程promise返回的值。在这种情况下，使用async()是很优雅的方法。

packaged_task提供了启动任务线程的简单方法。特别是它处理好了future和promise的关联关系，同时提供了包装代码以保证返回值/异常可以放到promise中，示例代码：

```
void comp(vector& v)
{
        // package the tasks:
        // (the task here is the standard
        //  accumulate() for an array of doubles):
        packaged_task pt0{std::accumulate};
        packaged_task pt1{std::accumulate};

        auto f0 = pt0.get_future();     // get hold of the futures
        auto f1 = pt1.get_future();

        pt0(&v[0],&v[v.size()/2],0);    // start the threads
        pt1(&[v.size()/2],&v[size()],0);

        return f0.get()+f1.get();       // get the results
}
```

参看：

- Standard: 30.6 Futures [futures]
- Anthony Williams:

  Moving Futures – Proposed Wording for UK comments 335, 336, 337 and 338.

  N2888==09-0078.

- Detlef Vollmann, Howard Hinnant, and Anthony Williams

  An Asynchronous Future Value (revised)

  N2627=08-0137.

- Howard E. Hinnant:

  Multithreading API for C++0X – A Layered Approach.

  N2094=06-0164. The original proposal for a complete threading package..

（翻译：interma）

垃圾回收（应用程序二进制接口）

在C++中，垃圾回收机制（自动回收没有被引用的内存区域）是可选的；也就是说在编译器中并不是一定要实现回收器。尽管如此，C++0x还是定义了垃圾回收器的功能。与此同时，C++0x还提供了应用程序二进制接口（ABI: Application Binary Interface）来辅助控制垃圾回收器的行为。

我们用“safely derived pointer”（3.7.3.3）（译注：我搜索后发现，是在3.7.3.3而不是3.7.4.3讲了safely derived pointer。这里可能是原文作者的笔误）来表示指针和生存时间的规则；粗略地说就是 “指向由new分配的对象或者指向相应的子对象的指针”。 下面是一些关于“not safely derived pointers”又名“disguised pointers”，或者说是为便于正常人理解和认可你的程序从而你应该注意的一些问题。

- 将指针暂时指到别处：

```
int* p = new int;
p+=10;
//…垃圾回收器可能在这里运行…
p-=10;
//在此，我们是否可以肯定开始为p分配的那块int内存还存在？
*p = 10;
```

- 将指针隐藏到一个int变量中：

```
int* p = new int;
int x = reinterpret_cast(p); // non-portable (不可移植)
p=0;
//…垃圾回收器可能在这里运行…
p = reinterpret_cast(x);
//在此，我们是否可以肯定开始为p分配的那块int内存还存在？
*p = 10;
```

- 还有更多甚至更危险的陷阱。比如I/O，以及将存储不同的数据位打散，..

虽然我们也有一些合理的理由来伪装指针（例如：xor有可能在exceptionally memory-constrained applications中导致错误），但是理由并不像一些程序员所认为的那么多。

程序员可以在代码中声明哪些地方不会有指针被发现（比如在一个图像中），也可以声明哪些内存区域不能被回收，即使垃圾回收器发现没有任何指针指向这块内存区域。以下是相关的例子：

void declare_reachable(void* p); //以p起始的内存区域

//（用能够记住所分配内存区域大小

//的内存分配操作符分配）不能被回收

```
template<class T> T* undeclared_reachable(T* p);

 void declare_no_pointers(char* p, size_t n);        //p[0..n] 中没有指针
 void undeclared_no_poitners(char* p, size_t n);
```

程序员要能够查询到关于指针安全和回收的规则是否是强制性的：

```
enum class pointer_saftety {relaxed, preferred, strict};

pointer_safety get_pointer_safety();
```

3.7.4.3[4]：满足以下三个条件的行为没有被定义：如果一个非”safely-derived pointer”值被释放，并且它所引用的对象是动态存储期（由new操作符动态创建并由delete销毁的对象拥有动态存储期），与此同时这个指针之前也没被声明为可到达的（20.7.13.7）。

- relaxed: safely-derived pointer和not safely-derived pointer被认为是等同的。这和C以及C++98中是一样的。但这并不是我的初衷。我的想法是用户如果没有使用有效的指针指向一个对象则应启用垃圾回收。
- Preferred：和relaxed类型一样。只不过垃圾回收器可能被用作内存泄露检测以及（或者）检测对象是否被一个错误的指针解引用。
- strict: safely-derived和not safely-derive这两种指针可能不再被等同。也就是说，垃圾回收器可能被启用而且将会忽略那些not safely derived pointer。

并不存在任何标准化的方法以供你选择任何一种。这是一个实现的质量（quality of implementation）和编程环境（programming environment）的问题。

另外，可以参考以下文献：

- the C++ draft 3.7.4.3
- the C++ draft 20.7.13.7
- Hans Boehm's

  GC page

- Hans Boehm's

  Discussion of Conservative GC

- final proposal
- Michael Spertus and Hans J. Boehm:

  The Status of Garbage Collection in C++0X

  .

  ACM ISMM'09.

（翻译：Yibo Zhu）

无序容器（unordered containers）

一个无序容器实际上就是某种形式的蛤希表。C++0x提供四种标准的无序容器：

- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

实际上，它们应该被称为hash_map等。但是因为有很多地方已经在使用hash_map这样的名字了，为了保证其兼容性，标准委员会不得不选择新的名字。而unordered_map是我们所能够找到的最好的名字了。无序（“unordered”）代表着map和unordered_map之间一个最本质的差别：当你使用map容器的时候，容器中的所有元素都是通过小于操作（默认情况下使用“`<`”操作符）排好序的，但是unordered_map并没有对元素进行排序，所以它并不要求元素具有小于操作符。并且，一个哈希表也并添言地提供排序的功能。相反，map容器中的元素也并不要求具有哈希函数。 基本上，当代码的优化是可能的并且我们有理由对其进行优化时，我们可以把unordered_map当作一个优化之后的map容器来使用。例如：

```
map<string,int> m {
    {“Dijkstra”,1972}, {“Scott”,1976},
    {“Wilkes”,1967}, {“Hamming”,1968}
};
m["Ritchie"] = 1983;
for(auto x : m)
    cout << ‘{‘ << x.first << ‘,’ << x.second << ‘}’;

// 使用优化之后的unordered_map
unordered_map<string,int> um {
    {“Dijkstra”,1972}, {“Scott”,1976},
    {“Wilkes”,1967}, {“Hamming”,1968}
};
um["Ritchie"] = 1983;
for(auto x : um)
    cout << ‘{‘ << x.first << ‘,’ << x.second << ‘}’;
```

map容器m的迭代器将以字母的顺序访问容器中的所有元素，而unordered_map容器um则并不按照这样的顺序（除非通过一些特殊的操作）。m和um两者的查找功能的实现机制是非常不同的。对于m，它使用的是复杂度为log2(m.size())的小于比较，而um只是简单地调用了一个哈希函数和一次或多次的相等比较。如果容器中的元素比较少（比如几打），很难说哪一种容器更快，但是对于大量数据（比如数千个）而言，unordered_map容器的查找速度要比map容器快很多。

更多内容稍后提供。

参考：

? Standard: 23.5 Unordered associative containers.

(翻译：Yibo Zhu)

锁（locks）

metaprogramming（元编程）and type traits

（译注：原作者还没有完成这个小节，等原作者完成后中文版随后奉上。这里补充一点关于元编程(metaprogramming)）的基础知识，供大家参考

1、何谓“元编程(Metaprogramming)”?

具备如下特征之一的程序编写称为元编程：

利用或者编写其它语言程序来作为所编写程序的数据

在程序运行时完成一些编程工作，而不是在编译时—具备这种特征的语言，我们也常成为动态语言

2、元编程的优势

元编程为程序开发人员提供了在与手动编写所有代码相同时间内，完成更多工作的可能，同时，由于

在面对新情况时，不需要重新编译，因此元编程为程序提供了更大的灵活性和可用性。

3、元编程语言

用于元编程的语言，称作元语言，被元语言所利用的语言通常称作对象语言。元语言具有自反性。所谓

自反性就是自己描述自己的特性。自反性是元编程中非常重要特点。有些语言中，将自身作为First-class object

也是非常有用的。对于有些可以调用元编成机制的程序语言，也具备自反性。

4、元编程方式

一般来说，实现元编程有两种方式。

第一种方法是调用API将运行时引擎中的代码展示为程序代码

第二种方法是动态执行包含在程序中的字符表达式，也就是我们常说的“程序生成程序”。

虽然，两种方法都可以实现元编程，但是大多数语言都只是支持其中的一种方法。

5、实例

1）bash script

```
#!/bin/bash # metaprogram echo ‘#!/bin/bash’ >program for ((I=1; I< =992; I++)) do echo "
```

该脚本产生993行代码，这便是用程序生成程序的一个例子

2）像Lisp， Python和Javascript等语言可以在程序运行期间修改或者增量编译，这些程序语言也可以在不产生新代码的

情况下进行元编程。

3）我们常见的编译器其实也是元编码的代表，编译器通常是用相对简约的高级语言来产生汇编或者机器代码。

（翻译：Yibo Zhu）

互斥

互斥是多线程系统中用于控制访问的一个原对象（primitive object）。下面的例子给出了它最基本的用法：

```
std::mutex  m;
int sh; //共享数据
// …
m.lock();
// 对共享数据进行操作：
sh += 1;
m.unlock();
```

在任何时刻，最多只能有一个线程执行到lock()和unlock()之间的区域（通常称为临界区）。当第一个线程正在临界区执行时，后续执行到m.lock()的线程将会被阻塞直到第一个进程执行到m.unlock()。这个过程比较简单，但是如何正确使用互斥并不简单。错误地使用互斥将会导致一系列严重后果。大家可以设想以下情形所导致的后果：一个线程只进行了lock()而没有执行相应unlock()；一个线程对同一个mutex对象执行了两次lock()操作；一个线程在等待unlock()操作时被阻塞了很久；一个线程需要对两个mutex对象执行lock()操作后才能执行后续任务。可以在很多书（译者注：通常操作系统相关书籍中会讲到）中找到这些问题的答案。在这里（包括Locks section一节）所给出的都是一些入门级别的。

除了lock()，mutex还提供了try_lock()操作。线程可以借助该操作来尝试进入临界区，这样一来该线程不会在失败的情况下被阻塞。下面例子给出了try_lock()的用法：

```
std::mutex m;
int sh; //共享数据
// …
if (m.try_lock()) {
    //操作共享数据
    sh += 1;
    m.unlock();
}
else {
    //可能在试图进入临界区失败后执行其它代码
}
```

recursive_mutex是一种能够被同一线程连续锁定多次的mutex。下面是recursive_mutex的一个实例：

```
std::recursive_mutex m;
 int sh; //共享数据
 //..
 void f(int i)
 {
     //…
     m.lock();
     //对共享数据进行操作
     sh += 1;
     if (-i>0) f(i);  //注意：这里对f(i)进行了递归调用，
     //将导致在m.unlock()之前多次执行m.lock()
     m.unlock();
     //…
 }
```

对于这点，我曾经夸耀过并且用f()调用它自身。一般地，代码会更加微妙。这是因为代码中经常会有间接递归调用。比如f()调用g()，而g()又调用了h()，最后h()又调用了f()，这样就形成了一个间接递归。

如果我想在未来的10秒内进入到一个mutex所划定的临界区，该如果实现？ timed_mutex类可以解决这个问题。事实上，关于它的使用可以被看做是关联了时间限制的try_lock()的一个特例。

```
std::timed_mutex m;
int sh; //共享数据
//…
if ( m.try_lock_for(std::chrono::seconds(10))) {
//对共享数据进行操作
sh += 1;
m.unlock();
}
else {
    //进入临界区失败，在此执行其它代码
}
```

try_lock_for()的参数是一个用相对时间表示的duration。如果你不想这么做而是想等到一个固定的时间点：一个time_point，你可以使用try_lock_until()：

```
std::timed_mutex m;
int sh; //共享数据
// …
if ( m.try_lock_until(midnight)) {
//对共享数据进行操作
sh += 1;
m.unlock();
}
else {
    //进入临界区失败，在此执行其它代码
}
```

这里使用midnight是一个冷笑话：对于mutex级别的操作，相应的时间是毫秒级别的而不是小时。

当然地，C++0x中也有recursive_timed_mutex。

mutex可以被看做是一个资源（因为它经常被用来代表一种真实的资源），并且当它对至少两个线程可见时它才是有用的。必然地，mutex不能被复制或者移动（正如你不能复制一个硬件的输入寄存器）。

令人惊讶地，实际中经常很难做到lock()s与unlock()s的匹配。设想一下那些复杂的控制结构，错误以及异常，要做到匹配的确比较困难。如果你可以选择使用locks去管理你的互斥，这将为你和你的用户节省大量的时间，再也不用熬夜通宵彻夜无眠了。（that will save you and your users a lot of sleep？？）。

同时可参考：

- Standard: 30.4 Mutual exclusion [thread.mutex]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.:

  Multi-threading Library for Standard C++ (Revision 1)

- ???

（翻译：Yibo Zhu）

随机数的产生

随机数有着广泛的用途。比如测试、游戏、仿真以及安全等领域都需要用到随机数。标准库所提供的多种可供选择的随机数产生器也恰恰反应了随机数应用范 围的多样性。随机数产生器由引擎（engine）和分布（distribution）两部分组成。其中，engine用于产生一个随机数序列或者伪随机数 序列；distribution则将这些数值映射到位于固定范围的某一数学分布中。关于分布的例子有：unifrom_int (所有的整数倍都被以相等的概率产生)以及normal_distribution (分布的概率密度函数曲线呈钟形)；每一种分布都处于某一特定的范围之内。例如：

```
//distribution将产生的随机数映射到整数1..6
uniform_int_distribution<int> one_to_six {1,6};

default_random_engine re {};        //默认的engine
```

如果想获得一个随机数，你可以用一个随机引擎为参数调用distribution来产生一个随机数：

```
int x = one_to_six(re); // x 是 [1:6]这个范围内的一个随机数
```

在每次调用的时候都需要提供一个引擎作为参数非常繁琐，所以我们可将引擎和 distribution邦定成一个函数对象，然后直接通过这个函数对象的调用来产生随机数，而不用每次调用都提供参数了。

```
auto dice {bind(one_to_six,re)}; //  产生一个新的随机数生成器
int x = dice(); // 调用dice函数对象，x是一个分布在 [1:6]范围的随机数
```

多亏在设计它是对一般性和性能的关注。在这方面，一位专家曾在评价标准库中随机数模块时说道：“在扩充的过程中，每一个随机数库想变成什么”。然而，它很 难真正让一个新手感觉到容易上手。在性能方面，我从没有见过随机数的接口成为性能的瓶颈。另外，我也一直会使用一个简单的随机数生器来教新手（具有一定的 基础）。下面的就是这样的一个可以说明问题的例子。

```
int rand_int(int low, high);   //按照均匀分布在区间[low: high]中产生一个随机数
```

然而我们如何实现rand_int()？我们必须在rand_int()中使用dice()之类的函数：

```
int rand_int(int low, int high)
{
     static default_random_engine re {};
     using Dist = uniform_int_distribution<int>;
     static Dist uid {};
     return uid(re, Dist::param_type{low,high});
}
```

关于rand_int()的定义依然是属于“专家级”的，但是应该把关于它的使用安排在C++课程的第一周。

在这里，我们举一个不太琐碎的关于随机数生成器的例子。这个例子中代码的功能是生成和打印一个正态分布。

```
default_random_engine re;   //默认引擎

normal_distribution<double> nd(31 /* mean */,
      8 /* sigma */);

auto norm = std::bind(nd, re);

vector<int> mn(64);

int main()
{
    for (int i = 0; i<1200; ++i)
               ++mn[round(norm())]; // 产生随机数

    for (int i = 0; i<mn.size(); ++i)
           {
        cout << i << '\t';
        for (int j=0; j<mn[i]; ++j)
                    cout << '*';

                cout << '\n';
    }
}
```

我运行了一个支持boost::random的版本并把它编辑到C++0x中，然后得到了下面的结果。

```
0
1
2
3
4 *
5
6
7
8
9 *
10 ***
11 ***
12 ***
13 *****
14 *******
15 ****
16 **********
17 ***********
18 ****************
19 *******************
20 *******************
21 **************************
22 **********************************
23 **********************************************
24 ********************************************
25 *****************************************
26 *********************************************
27 *********************************************************
28 ***************************************************
29 ******************************************************************
30 **********************************************
31 *********************************************************************
32 **********************************************
33 *************************************************************
34 **************************************************************
35 ***************************************
36 ***********************************************
37 **********************************************
38 *********************************************
39 ********************************
40 ********************************************
41 ***********************
42 **************************
43 ******************************
44 *****************
45 *************
46 *********
47 ********
48 *****
49 *****
50 ****
51 ***
52 ***
53 **
54 *
55 *
56
57 *
58
59
60
61
62
63
```

另外，可以参考以下文献：

- Standard 26.5: Random number generation

正则表达式（regular expressions）

抱歉，本主题尚未完成，请稍后再来。

（翻译：Yibo Zhu）

具有作用域的内存分配器

为了使容器对象小巧和简单起见，C++98没有要求容器支持具有状态的内存分配器，即不用把分配器对象存储在容器对象中。在C++11中，这仍然默认做法。但是，在C++0x中，也可以使用具有状态的内存分配器：这种内存分配器拥有一个指向分配区域的指针。例如：

```
template<class T> class Simple_alloc {   // C++98 style
   // no data
   // usual allocator stuff
};

class Arena {
   void* p;
   int s;
public:
   Arena(void* pp, int ss);
   // allocate from p[0..ss-1]
};

template<class T> struct My_alloc {
   Arena& a;
   My_alloc(Arena& aa) : a(aa) { }
   // usual allocator stuff
};

Arena my_arena1(new char[100000],100000);
Arena my_arena2(new char[1000000],1000000);

vector<int> v0;   // allocate using default allocator

vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1});   // allocate from my_arena1

vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2});   // allocate from my_arena2

vector<int,Simple_alloc<int>> v3;   // allocate using Simple_alloc
```

通常我们可以使用typedef来简化上述例子中繁冗的表达。

虽然并不能保证默认内存分配器和Simple_alloc不在一个vector对象中占用空间，但是在实现库的时候可以使用一些模板的元程序设计方法来做到这点。因此，如果对象拥有状态的话（比如My_alloc），内存分配器将会带来一定的空间开销。

在同时使用容器和用户自定义内存分配器时存在一个更为隐蔽的问题：一个内存分配器成员是否应该和它的容器处于相同的分配区域中？例如，你使用

Your_allocator来给Your_string的成员分配内存，而我用My_allocator来给My_vector的成员分配内存。在这种

情况下，My_vector会使用哪个分配器。要解决这问题的话就需要告诉容器使用什么分配器来传递成员。下面的例子将给出实现这一点的方法。在这个例子

中我将使用分配器My_alloc来分配vector成员和string成员。首先，我必须要有一个能够接受My_alloc对象的string版本：

```
//使用My_alloc的一个string类型
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
```

然后，我要有一个能够接受这种string类型以及My_alloc对象的vector版本。同时，该vector版本需要能够把My_alloc对象传递给string。

```
using svec =  vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
```

最后，我们可以按照下面的示例实现My_alloc类型的分配器：

```
svec v(svec::allocator_type(My_alloc{my_arena1}));
```

在此，svec是一个成员为string类型的vector，并且该vector使用My_alloc来为其string类型的成员分配内存。另外，标准

库中新提供了”adaptor” (“wrapper type”)

scoped_allocator_adaptor。它可以用来指明使用My_alloc来为string类型变量分配内存。需要注意的

是，scoped_allocator_adaptor可以将My_alloc转换为My_alloc。而这正是xstring所需要提供的功能。

于是，我们又有了4种可用于替换的方法：

```
// vector and string use their own (the default) allocator:
using svec0 = vector<string>;
svec0 v0;

// vector (only) uses My_alloc and string uses its own (the default) allocator:
using svec1 = vector<string,My_alloc<string>>;
svec1 v1(My_alloc<string>{my_arena1});

// vector and string use My_alloc (as above):
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using svec2 = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena1});

// vector uses My_alloc and string uses My_string_alloc:
using xstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
using svec3 = vector<xstring2,scoped_allocator_adaptor<My_alloc<xstring>, My_string_alloc
svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>, My_string_alloc<char>>{my_arena1,my
```

很明显，第一种方法svec0是最常用的。但是在内存会严重影响性能的系统中，其它版本（特别是svec2）将会显得尤为重要。当然了，我们可以使用一些

typedef来增强代码的可读性。不过还好，毕竟你不用每天都写这些。另外，我们提供了scoped_allocator_adaptor2的一个变

种：scoped_allocator_adaptor2。它用于两个非默认内存分配器不一样的情况。

同时可参考：

- Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]
- Pablo Halpern:

  The Scoped Allocator Model (Rev 2).

  N2554=08-0064.

（翻译：Yibo Zhu）

共享资源的智能指针——shared_ptr

shared_ptr被用来表示共享的拥有权。也就是说，当两段代码都需要访问一些数据，而它们又都没有独占该数据的所有权（从某种意义上来说就是该段代码负责销毁该对象）。这是我们就需要shared_ptr。shared_ptr是一种计数指针。当引用计数变为0时，shared_ptr所指向的对象就会被删除。下面我们用一段代码来说明这点。

```
void test()
{
    shared_ptr p1(new int);    // 计数是1
    {
        shared_ptr p2(p1);    //计数是2
        {
            shared_ptr p3(p1);    // 计数是3
        }    //计数变为2
    } //计数变为1
}    // 在此，计数变为0。同时int对象被删除
```

现在来看一个更为实际的例子。在这个例子中，我们用指针指向图中的节点。一个需要解决的问题是当从一个节点上移除一个指针时并不知道时候还有其它指针指向这个节点。如果节点能够拥有一些资源，从而需要析构器采取一些行动（一个典型的例子就是文件句柄。当节点被检测到时，文件句柄相应的文件就会被关闭）。这样以来，通过shared_ptr就可以解决这个问题。你可以认为使用shared_ptr的目的和你使用垃圾回收器的目的是一样的。只是出处于经济性的考虑，你没有足够的垃圾，或者执行环境不允许那么做，或者所管理的资源不仅仅是内存（如文件句柄）。例如：

```
struct Node {    // 注意：其它的节点也可能指向该节点
    shared_ptr left;
    shared_ptr right;
    File_handle f;
    // …
};
```

这里Node的析构器（隐式的析构器即可）删除了它的子节点。也就是说Node的析构器调用了left和right的析构器。因为left是一个shared_ptr，所以当left是最后一个指向该Node的指针时，该节点将会被删除。处理right的方式和left的类似。f的析构器将按照f的要求执行。

需要注意的是，当仅需要将一个指针从一个拥有者传个另一个时，你不应使用shared_ptr。这是unique_ptr的用途。unique_ptr会以更为小的开销来更好的实现这个功能。如果你曾经使用计数指针作为工厂函数的返回值或者类似的情形，可以考虑升级使用unique_ptr而不是shared_ptr。

另外，不要不加思考地把指针替换为shared_ptr来防止内存泄露。shared_ptr并不是万能的，而且使用它们的话也是需要一定的开销的：

- 环状的链式结构shared_ptr将会导致内存泄露（你需要一些逻辑上的复杂化来打破这个

环。比如使用weak_ptr）。
- 共享拥有权的对象一般比限定作用域的对象生存更久。从而将导致更高的平均资源使用时间。
- 在多线程环境中使用共享指针的代价非常大。这是因为你需要避免关于引用计数的数据竞争。
- 共享对象的析构器不会在预期的时间执行。
- 与非共享对象相比，在更新任何共享对象时，更容易犯算法或者逻辑上的错误。

shared_ptr用于表示共享拥有权。然而共享拥有权并不是我的初衷。在我看来，一个更好的办法是为对象指明拥有者并且为对象定义一个可以预测的生存范围。

同时可参考：

- the C++ draft: Shared_ptr (20.7.13.3)

（翻译：Yibo Zhu）

smart pointers

线程（thread）

线程（译注：大约是C++11中最激动人心的特性了）是一种对程序中的执行或者计算的表述。跟许多现代计算一样，C++11中的线程之间能够共享地址空间。从这点上来看，它不同于进程：进程一般不会直接跟其它进程共享数据。在过去，C++针对不同的硬件和操作系统有着不同的线程实现版本。如今，C++将线程加入到了标准件库中：一个标准线程ABI。

许多大部头书籍以及成千上万的论文都曾涉及到并发、并行以及线程。在这一条FAQ里几乎不涉及这些内容。事实上，要做到清楚地思考并发非常难。如果你想编写并发程序，请至少看一本书。不要依赖于一本手册、一个标准或者一条FAQ。

在用一个函数或者函数对象（包括lambda）构造std::thread时，一个线程便启动了。

```
#include <thread>

void f();

struct F {
    void operator()();
};

int main()
{
    std::thread t1{f};    // f() 在一个单独的线程中执行
    std::thread t2{F()};    // F()() 在一个单独的线程中执行
}
```

然而，无论f()和F()执行任何功能，都不能给出有用的结果。这是因为程序可能会在t1执行f()之前或之后以及t2执行F()之前或之后终结。我们所期望的是能够等到两个任务都完成，这可以通过下述方法来实现：

```
int main()
{
    std::thread t1{f};    // f() 在一个单独的线程中执行
    std::thread t2{F()};    // F()()在一个单独的线程中执行

    t1.join();    // 等待t1
    t2.join();    // 等待t2
}
```

上面例子中的join()保证了在t1和t2完成后程序才会终结。这里”join”的意思是等待线程返回后再终结。

通常我们需要传递一些参数给要执行的任务。例如：

```
void f(vector<double>&);

struct F {
vector<double>& v;
F(vector<double>& vv) :v{vv} { }
void operator()();
};

int main()
{
    // f(some_vec) 在一个单独的线程中执行
    std::thread t1{std::bind(f,some_vec)};

    // F(some_vec)() 在一个单独的线程中执行
    std::thread t2{F(some_vec)};

    t1.join();
    t2.join();
}
```

上例中的标准库函数bind会将一个函数对象作为它的参数。

通常我们需要在执行完一个任务后得到返回的结果。对于那些简单的对返回值没有概念的，我建议使用std::future。另一种方法是，我们可以给任务传递一个参数，从而这个任务可以把结果存在这个参数中。例如：

```
void f(vector<double>&, double* res);    // 将结果存在res中

struct F {
    vector<double>& v;
    double* res;
    F(vector<double>& vv, double* p) :v{vv}, res{p} { }
    void operator()();    //将结果存在res中
};

int main()
{
    double res1;
    double res2;

    // f(some_vec,&res1) 在一个单独的线程中执行
    std::thread t1{std::bind(f,some_vec,&res1)};
    // F(some_vec,&res2)() 在一个单独的线程中执行
    std::thread t2{F(some_vec,&res2)};

    t1.join();
    t2.join();

    std::cout << res1 << " " << res2 << '\n';
}
```

但是关于错误呢？如果一个任务抛出了异常应该怎么办？如果一个任务抛出一个异常并且它没有捕获到这个异常，这个任务将会调用std::terminate()。调用这个函数一般意味着程序的结束。我们常常会为避免这个问题做诸多尝试。std::future可以将异常传送给父线程（这正是我喜欢future的原因之一）。否则，返回错误代码。

除非一个线程的任务已经完成了，当一个线程超出所在的域的时候，程序会结束。很明显，我们应该避免这一点。

没有办法来请求（也就是说尽量文雅地请求它尽可能早的退出）一个线程结束或者是强制（也就是说杀死这个线程）它结束。下面是可供我们选择的操作：

- 设计我们自己的协作的中断机制（通过使用共享数据来实现。父线程设置这个数据，子线程检查这个数据（子线程将会在该数据被设置后很快退出））。
- 使用thread::native_handle()来访问线程在操作系统中的符号
- 杀死进程（std::quick_exit()）
- 杀死程序(std::terminate())

这些是委员会能够统一的所有的规则。特别地，来自POSIX的代表强烈地反对任何形式的“线程取消”。然而许多C++的资源模型都依赖于析构器。对于每种系统和每种可能的应有并没有完美的解决方案。

线程中的一个基本问题是数据竞争。也就是当在统一地址空间的两个线程独立访问一个对象时将会导致没有定义的结果。如果一个（或者两个）对对象执行写操作，而另一个（或者两个）对该对象执行读操作，两个线程将在谁先完成操作方面进行竞争。这样得到的结果不仅仅是没定义的，而且常常无法预测最后的结果。为解决这个问题，C++0x提供了一些规则和保证从而能够让程序员避免数据竞争。

- C++标准库函数不能直接或间接地访问正在被其它线程访问的对象。一种例外是该函数通过参数（包括this）来直接或间接访问这个对象。
- C++标准库函数不能直接或间接修改正在被其它线程访问的对象。一种例外是该函数通过非const参数（包括this）来直接或间接访问这个对象。
- C++标准函数库的实现需要避免在同时修改统一序列中的不同成员时的数据竞争。

除非已使用别的方式做了声明，多个线程同时访问一个流对象、流缓冲区对象，或者C库中的流可能会导致数据竞争。因此除非你能够控制，绝不要让两个线程来共享一个输出流。

你可以

- 等待一个线程一定的时间
- 通过互斥来控制对数据的访问
- 通过锁来控制对数据的访问
- 使用条件变量来等待另一个线程的行为
- 通过future来从线程中返回值

同时可参考：

- Standard: 30 Thread support library [thread]
- 17.6.4.7 Data race avoidance [res.on.data.races]
- ???
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.:

  Multi-threading Library for Standard C++ (Revision 1)

N2497==08-0007

- H.-J. Boehm, L. Crowl:

  C++ object lifetime interactions with the threads API

  N2880==09-0070.

- L. Crowl, P. Plauger, N. Stoughton:

  Thread Unsafe Standard Functions

  N2864==09-0054.

- WG14:

  Thread Cancellation N2455=070325.

（翻译：Yibo Zhu）

时间工具程序

在编写程序时，我们常常需要定时执行一些任务。例如，标准库mutexes和locks提供了的一些选项就需要这一定时功能：线程等待一段时间(duration)或者等到某一给定时刻(time_point)。

如果你需要得到当前时刻，你可以调用system_clock、monotonic_clock、high_resolution_clock中任何一个时钟的now()方法。例如：

```
monotonic_clock::time_point t = monotonic_clock::now();
// 执行一些代码
monotonic_clock::duration d = monotonic_clock::now() – t;
// 一些需要d个单位时间的任务
```

在上面例子中，一个时钟返回一个time_point和一个duration。其中duration是该时钟它返回的两个time_point的差值。如果你对细节不感兴趣，你可以使用auto类型。

```
auto t = monotonic_clock::now();
// 执行一些代码
auto d = monotonic_clock::now() – t;
// 一些需要d个单位时间的任务
```

这里提供的时间工具是为了高效支持系统内部的应用。它们不会提供便捷的工具来帮助你维护你的社交日历。事实上，这些时间工具源自于高能物理对时间度量的高精度要求。为了能够表达所有的时间尺度（比如世纪和皮秒），同时避免单位、打字排版以及舍入时的混淆，使用编译时的有理数包来表示duration和time_point。一个duration由两部分组成：一个数字时钟”tick”（滴答）和能够表示一个tick期望(一秒还是一毫秒？)的事物（一个period）。这里的period是duration类型的一部分。下面的表格摘自标准头文件中，它定义了国际单位系统中的period。这或许会帮助你明白它们的使用范围。

```
// 为方便起见，对国际单位做的typedef:
typedef ratio<1, 1000000000000000000000000> yocto;  // 有条件的支持
typedef ratio<1,    1000000000000000000000> zepto;  // 有条件的支持
typedef ratio<1,       1000000000000000000> atto;
typedef ratio<1,          1000000000000000> femto;
typedef ratio<1,             1000000000000> pico;
typedef ratio<1,                1000000000> nano;
typedef ratio<1,                   1000000> micro;
typedef ratio<1,                      1000> milli;
typedef ratio<1,                       100> centi;
typedef ratio<1,                        10> deci;
typedef ratio<                       10, 1> deca;
typedef ratio<                      100, 1> hecto;
typedef ratio<                     1000, 1> kilo;
typedef ratio<                  1000000, 1> mega;
typedef ratio<               1000000000, 1> giga;
typedef ratio<            1000000000000, 1> tera;
typedef ratio<         1000000000000000, 1> peta;
typedef ratio<      1000000000000000000, 1> exa;
typedef ratio<   1000000000000000000000, 1> zetta;  //有条件的支持
typedef ratio<1000000000000000000000000, 1> yotta;  //有条件的支持
```

编译时有理数提供的常用算术操作符(`+` , `-` , `*` , and `/`)和比较操作符 (`==` , `!=` , `<` , `<=` , `>` , `>=`)适用于合理的duration和time_point组合（比如你不能对两个time_point进行加法运算）。系统会对这些运算进行溢出以及除数为0的检查。由于这是编译时的工具，所以不用担心它在的运行时的性能。另外，你还可以使用++、–、+=、-=以及/=来操作duration，同时可以使用tp+=d和tp-=d来操作time_point tp和duration d。

下面是一些使用定义在中的duration类型的例子：

```
microseconds mms = 12345;
milliseconds ms = 123;
seconds s = 10;
minutes m = 30;
hours h = 34;

auto x = std::chrono::hours(3);    // 显式使用命名空间
auto x = hours(2)+minutes(35)+seconds(9); // 假设合适的"using"
```

你不能用一个分数来初始化duration。比如，不要用2.5秒，而应该用2500毫秒。这是因为duration被解释为若干个tick，而每个tick表示一个duration时间段的单位，比如上面例子中所定义的milli和kilo。所以必须用整数来初始化。Duration的默认单位是秒。也就是说，时间段为1的tick被解释为1秒。在程序中，我们也可以指明duration的单位。

```
duration d0 = 5;        // 秒 (默认值)
duration d1 = 99;        // 千秒
duration > d2 = 100;    // d1和d2的类型相同
```

如果我们想利用duration来做一些事（比如打印出这个duration的值），那么我们必须给出它的单位。如：分钟或者微妙。例如：

```
auto t = monotonic_clock::now();
// 执行一些代码
nanoseconds d = monotonic_clock::now() – t;     // 我们希望结果的单位是纳秒
cout << “something took ” << d << “nanosecondsn”;
```

或者，我们可以将duration转换成一个浮点数

```
auto t = monotonic_clock::now();
//执行一些代码
auto d = monotonic_clock::now() – t;
cout << “something took ” << duration(d).count() << “secondsn”;
```

这里的count()返回tick的数量。

同时可参考：

- Standard: 20.9 Time utilities [time]
- Howard E. Hinnant,

  Walter E. Brown,

  Jeff Garland,

  and Marc Paterno:

  A Foundation to Sleep On.

  N2661=08-0171.

  Including “A Brief History of Time” (With apologies to Stephen Hawking).

(翻译：Yibo Zhu)

标准库中的元组（std::tuple）

在标准库中，tuple（一个N元组：N-tuple）被定义为N个值的有序序列。在这里，N可以是从0到文件中所定义的最大值中的任何一个常数。你可以认为tuple是一个未命名的结构体，该结构体包含了特定的tuple元素类型的数据成员。特别需要指出的是，tuple中元素是被紧密地存储的（位于连续的内存区域），而不是链式结构。

可以显式地声明tuple的元素类型，也可以通过make_tuple()来推断出元素类型。另外，可以使用get()来通过索引（和C++的数组索引一样，从0而不是从1开始）来访问tuple中的元素。

```
tuple<string,int> t2("Kylling",123);

// t的类型被推断为tuple
auto t = make_tuple(string("Herring"),10, 1.23);
// 获取元组中的分量
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

有时候，我们需要一个编译时异构元素列表（a heterogeneous list of elements at compile time），但又不想定义一个有名字的结构来保存。这种情况下，我们就可以使用tuple（直接地或间接地）. 例如，我们在std::function and std::bind中使用tuple来保存参数。

最常用的tuple是2-tuple(二元组)，也就是一个pair。但是标准库已经定义了pair：std::pair (20.3.3 Pairs)。 我们可以使用pair来初始化一个tuple，然而反之则不可。

另外，需要为tuple中的元素类型定义比较操作（`==`，`!=`，`<`，`<=`，`>`，和 `>=`）.

参考：

- Standard: 20.5.2 Class template tuple
- Variadic template paper
- Boost::tuple

unique_ptr

unique_ptr（定义在中）提供了一种严格的语义上的所有权

- 拥有它所指向的对象
- 无法进行复制构造，也无法进行复制赋值操作（译注：也就是对其无法进行复制，我们无法得到指向同一个对象的两个unique_ptr），但是可以进行移动构造和移动赋值操作
- 保存指向某个对象的指针，当它本身被删除释放的时候（例如，离开某个作用域），会使用给定的删除器（deleter）删除释放它指向的对象，

unique_ptr的使用能够包括：

- 为动态申请的内存提供异常安全
- 将动态申请内存的所有权传递给某个函数
- 从某个函数返回动态申请内存的所有权
- 在容器中保存指针

“所有auto_ptr应该已经具有的（但是我们无法在C++98中实现的）功能”

unique_ptr十分依赖于右值引用和移动语义。

下面是一段传统的会产生不安全的异常的代码：

```
X* f()
{
    X* p = new X;
    // 做一些事情 – 可能会抛出某个异常
    return p;
}
```

解决方法是，用一个unique_ptr 来管理这个对象的所有权，由其进行这个对象的删除释放工作：

```
X* f()
{
    unique_ptr p(new X); // 或者使用{new X}，但是不能 = new X
    // 做一些事情 – 可能会抛出异常
    return p.release();
}
```

现在，如果程序执行过程中抛出了异常，unique_ptr就会（毫无疑问地）删除释放它所指向的对象，这是最基本的RAII。但是，除非我们真的需要返回一个内建的指针，我们可以返回一个unique_ptr，让事情变得更好。

```
unique_ptr f()
{
    unique_ptr p(new X); // 或者使用{new X}，但是不能 = new X
    //做一些事情 – 可能会抛出异常
    return p; // 对象的所有权被传递出f()
}
```

现在我们可以这样使用函数f()：

```
void g()
{
    unique_ptr q = f(); // 使用移动构造函数（move constructor）
    q->memfct(2); // 使用q
    X x = *q; // 复制指针q所指向的对象
    // …
} // 在函数退出的时候，q以及它所指向的对象都被删除释放
```

unique_ptr拥有“移动意义（move semantics）”，所以我们可以使用函数f() 返回的右值对q进行初始化，这样就简单地将所有权传递给了q。

在那些要不是为了避免不安全的异常问题（以及为了保证指针所指向的对象都被正确地删除释放），我们不可以使用内建指针的情况下，我们可以在容器中保存unique_ptr以代替内建指针：

```
vector<unique_ptr<string>> vs { new string{“Doug”},
                                new string{“Adams”} };
```

unique_ptr可以通过一个简单的内建指针构造完成，并且与内建指针相比，两者在使用上的差别很小。特殊情况下，unique_ptr并不提供任何形式的动态检查(?)。

参考：

- the C++ draft section 20.7.10
- Howard E. Hinnant: unique_ptr Emulation for C++03 Compilers
- final proposal.

weak_ptr

弱指针（weak pointer）经常被解释为用来打破使用shared_ptr管理的数据结构中循环(?)。但是我认为，将weak_ptr看成是指向具有下列特征的对象的指针更好一些。

- 只有当对象存在的时候，你才需要对其进行访问
- 并且它可能被其他人删除释放
- 并且在最后一次使用之后调用其析构函数（通常用于释放那些不具名的内存(anon-memory)资源

（译注：weak_ptr可以保存一个“弱引用”，指向一个已经用shared_ptr进行管理的对象。为了访问这个对象，一个weak_ptr可以通过shared_ptr的构造函数或者是weak_ptr的成员函数lock()转化为一个shared_ptr。当最后一个指向这个对象的shared_ptr退出其生命周期并且这个对象被释放之后，将无法从指向这个对象的weak_ptr获得一个shared_ptr指针，shared_ptr的构造函数会抛出异常，而weak_ptr::lock也会返回一个空指针。）

我们来考虑一下如何实现一个老式的“星盘棋”（ asteroid game）游戏，所有星星（asteroid）都属于游戏（the game），但是所有星星都必须与它周围的 星星保持联系，并且与之保持相反的状态。要维持一个相反的状态，通常会消去一个或者多个星星，也就是会调用其析构函数。每个星星必须有一个列表来保存记录它周围的星星。这里需要注意的是，在这样一个相邻星星列表中的星星不应该是具有完整生命的(?)，所以shared_ptr在这种情况下并不适合。另外一方面，当另外一个星星正看着某个星星时，（例如，依赖于这个星星计算其相反状态），这个星星就不能被析构。 当然，星星的析构函数必须被调用以释放其占用的资源（比如与图形系统的连接）。我们所需要的是一个在任何时间都应该保持完整无缺，并且随时都可以从中获取一个星星的星星列表(?)。weak_ptr可以帮我们做到这一切：

```
void owner()
{
    // …
    vector<shared_ptr<Asteroid>> va(100);
    for (int i=0; i<va.size(); ++i) { // 访问相邻的星星，计算相反状态
        va[i].reset(new Asteroid(weak_ptr(va[neighbor]));
        launch(i);
    }
    // …
}
```

reset() 可以让一个shared_ptr指向另外一个新的对象。

当然，我对ower类作了相当大的简化，并且只给了每个星星一个邻居。这里的关键是，我们使用了weak_ptr指向其邻居星星。在计算相反状态的时候，ower类则使用shared_ptr来代表星星与owner之间的所属关系(?)。一个星星的相反关系的计算应该是这样的：

```
void collision(weak_ptr<Asteroid> p)
{
    // p.lock返回一个指向p所指对象的shared_ptr
    if (auto q = p.lock()) {
        // … p以及q指向的星星对象依然存在:进行计算…
    }
    else {
        // … oops: 星星对象已经被析构，我们可以忘掉它了(?)…
    }
}
```

注意，即使owner决定关闭整个游戏并释放所有的星星对象（通过删除代表所属关系的多个shared_ptr），每一个正在计算过程中的星星对象仍然可以正确地结束，因为p.lock()将维持一个shared_ptr，直到计算过程结束。（译注，也即是说，如果正在计算过程中关闭游戏并通过shared_ptr释放对象，那么p.lock()会维持一个shared_ptr，这样可以使得shared_ptr不会变成0，在计算过程中，星星对象也就不会被错误地释放。当整个计算过程结束后，shared_ptr的引用计数变为0，星星对象被正确释放。）

我期望看到weak_ptr比简单的shared_ptr更少地被用到，并且我希望unique_ptr可以比shared_ptr更加流行，因为unique_ptr的所属关系更简单一些（译注：只能有一个 unique_ptr指针指向某个对象，不向shared_ptr,可以同时有多个shared_ptr指向同一个对象）并且性能更高，因而可以让局部的代码更容易理解。

参考

- the C++ draft: Weak_ptr (20.7.13.3)

system error